HARP: A MACHINE LEARNING FRAMEWORK ON TOP OF THE COLLECTIVE COMMUNICATION LAYER FOR THE BIG DATA SOFTWARE STACK

Bingjing Zhang

Submitted to the faculty of the University Graduate School
in partial fulfillment of the requirements
for the degree
Doctor of Philosophy
in the School of Informatics and Computing,
Indiana University
May 2017

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Doctoral Committee

Judy Qiu, Ph.D.

Geoffrey Charles Fox, Ph.D.

David J Crandall, Ph.D.

Paul Purdom, Ph.D.

March 27, 2017

Acknowledgments

I sincerely thank my advisor, Dr. Qiu, and my colleagues: Dr. Peng, Dr. Chen, Meng Li, Yiming Zou, Thomas Wiggins, and Allan Streib for their support to my research work. I also thank Dr. Crandall, Dr. Fox, and Dr. Purdom, my research committee members, for providing guidance in my thesis research. I am very grateful to my wife, Jane, and my parents for their love and support, which has been key to helping me complete my Ph.D. program.

Bingjing Zhang

HARP: A MACHINE LEARNING FRAMEWORK ON TOP OF THE COLLECTIVE COMMUNICATION LAYER FOR THE BIG DATA SOFTWARE STACK

Almost every field of science is now undergoing a data-driven revolution requiring analyzing massive datasets. Machine learning algorithms are widely used to find meaning in a given dataset and discover properties of complex systems. At the same time, the landscape of computing has evolved towards computers exhibiting many-core architectures of increasing complexity. However, there is no simple and unified programming framework allowing for these machine learning applications to exploit these new machines' parallel computing capability. Instead, many efforts focus on specialized ways to speed up individual algorithms. In this thesis, the Harp framework, which uses collective communication techniques, is prototyped to improve the performance of data movement and provides high-level APIs for various synchronization patterns in iterative computation.

In contrast to traditional parallelization strategies that focus on handling high volume training data, a less known challenge is that the high dimensional model is also in high volume and difficult to synchronize. As an extension of the Hadoop MapReduce system, Harp includes a collective communication layer and a set of programming interfaces. Iterative machine learning algorithms can be parallelized through efficient synchronization methods utilizing both inter-node and intra-node parallelism. The usability and efficiency of Harp's approach is validated on applications such as K-means Clustering, Multi-Dimensional Scaling, Latent Dirichlet Allocation and Matrix Factorization. The results show that these machine learning applications can achieve high parallel performance on Harp.

Judy Qiu, Ph.D.

Geoffrey Charles Fox, Ph.D.

David J Crandall, Ph.D.

Paul Purdom, Ph.D.

Contents

1 Introduction

Data analytics is undergoing a revolution in many scientific domains. Machine learning algorithms[1] have become popular methods for analytics, which allow computers to learn from existing data and make data-based predictions. They have been widely used in computer vision, text mining, advertising, recommender systems, network analysis, and genetics. Unfortunately, analyzing big data usually exceeds the capability of a single or even a few machines owing to the incredible volume of data available, and thus requires algorithm parallelization at an unprecedented scale. Scaling up these algorithms is challenging because of their prohibitive computation cost, not only the need to process enormous training data in iterations, but also the requirement to synchronize big models in rounds for algorithm convergence.

Through examining existing parallel machine learning implementations, I conclude that parallelization solutions can be categorized into four types of computation models: "Locking", "Rotation", "Allreduce", and "Asynchronous". My classification of the computation models is based on model synchronization patterns (synchronized algorithms or asynchronous algorithms) and the effectiveness of the model parameter update (the latest model parameters or stale model parameters). As a particular case study, I chose a representative machine learning algorithm, Collapsed Gibbs Sampling (CGS) [1, 2] for Latent Dirichlet Allocation (LDA) [3], to understand the differences between these computation models. LDA is a widely used machine learning technique for big data analysis. It includes an inference algorithm that iteratively updates a model until the model converges. The LDA application is commonly solved by the CGS algorithm. A major challenge is the scaling issue in parallelization owing to the fact that the model size is huge, and parallel workers need to communicate the model parameters continually. I identify three important features of the parallel LDA computation to consider here:

1. The volume of model parameters required for local computation is high.
2. The time complexity of local computation is proportional to the required model size.
3. The model size shrinks as it converges.

In LDA parallelization, compared with the "Asynchronous" computation model, the "Allreduce" computation model with optimized communication routing can improve model synchronization

[1] `http://ai.stanford.edu/~ronnyk/glossary.html`

speed, thus allowing the model to converge faster. This performance improvement derives not only from accelerated communication but also from reduced iteration computation time as the model size shrinks during convergence. The results also reveal that the "Rotation" computation model can achieve faster model convergence speed than the "Allreduce" computation model. The main rationale is that in the training procedure running on each worker, the "Allreduce" computation model uses stale model parameters, but the "Rotation" computation model uses the latest model parameters. The effect of the model update in convergence reduces when stale model parameters are used. Besides, even when two implementations use the "Rotation" computation model, communication through sending model parameters in chunks can further reduce communication overhead compared with the flooding of small messages.

Synchronized communication performed by all the parallel workers is referred to as "collective communication" in the High-Performance Computing (HPC) domain. In MPI [4], collective communication patterns are implemented with various optimizations and invoked as operations. Though the collective communication technique can result in efficient model synchronization as shown in LDA, it has not been thoroughly applied to many machine learning applications. However, MPI only provides basic collective communication operations which describe process-to-process synchronization patterns, so it does not cover all the complicated parameter-to-parameter synchronization patterns. In that case, users have to rely on send/receive calls to develop customized synchronization patterns. The applications developed achieve high performance but create a complicated code base.

Another way to implement machine learning algorithms is to use big data tools. Initially, many machine learning algorithms were implemented in MapReduce [5, 6]. However, these implementations suffer from repeated input data loading from the distributed file systems and slow disk-based intermediate data synchronization in the shuffling phase. This motivates the design of iterative MapReduce tools such as Twister [7] and Spark [8], which utilizes memory for data caching and communication and thus drastically improve the performance of large-scale data processing. Later, big data tools have expanded rapidly and form an open-source software stack. Their programming models are not limited to MapReduce and iterative MapReduce. In graph processing tools [9], input data are abstracted as a graph and processed in iterations, while intermediate data per iteration are expressed as messages transmitted between vertices. In Parameter Server type tools, model param-

eters are stored in a set of server machines, and they can be retrieved asynchronously in parallel processing. To support these tools, big data systems are split into multiple layers. A typical layered architecture is seen in the Apache Big Data Stack (ABDS) [10]. Though these tools are continually evolving and improving their performance, there are still fundamental issues unsolved. To simplify the programming process, many tools' design tries to fix the parallel execution flow, and developers are only required to fill the bodies of user functions. However, this results in limited support of synchronization patterns so that the parallelization performance suffers from improper usage of synchronization patterns and inefficient synchronization performance.

To solve all these problems in machine learning algorithm parallelization, in this thesis, I propose the Harp framework [11]. Its approach is to use optimized collective communication techniques to improve model synchronization performance. Harp provides high-level programming interfaces for various synchronization patterns in iterative machine learning computations, which are not well-supported in current big data tools. Therefore, a MapCollective programming model is extended from the original MapReduce programming model. The MapCollective model still reads Key-Value pairs as inputs. However, instead of using a shuffling phase, Harp uses collective communication operations on partitioned distributed datasets. All these Harp programming interfaces can be mapped to the computation models. Harp is designed as a plug-in to Hadoop[2] so that it can enrich the ABDS with HPC methods. With improved expressiveness and performance on synchronization, HPC-ABDS can support various machine learning applications.

With the Harp framework, I then focus on building a machine learning library with the MapCollective programming model. Several machine learning algorithms are investigated in this thesis. First, K-means [12] and SMACOF for Multi-Dimensional Scaling [13] are implemented with the "Allreduce" computation model. These two algorithms use the classic "allgather" and "allreduce" collective communication operations. Results on the Big Red II Super Computer[3] show that by applying efficient routing algorithms in collective communication, Harp can achieve high speedup. Three other algorithms, CGS for LDA, Stochastic Gradient Descent (SGD) for Matrix Factorization (MF) [14], and Cyclic Coordinate Descent (CCD) for MF [15], are implemented using the "Rotation" computation model. These algorithms are implemented with further abstracted high-level

[2] `http://hadoop.apache.org`
[3] `https://kb.iu.edu/d/bcqt`

programming interfaces. Pipelined rotation is used to reduce synchronization overhead. Dynamic rotation control is applied to CGS for LDA and SGD for MF in order to improve load balancing. The performance results on Intel Haswell and Knights Landing clusters[4] show that the Harp solution achieves faster model convergence speed and higher scalability than previous work.

To summarize, this thesis makes the following seven contributions:

1. Identify model-centric parallelization as the key to parallel machine learning and categorize algorithm implementations into four computation models
2. Identify algorithm features and model synchronization patterns of the LDA application and execute comparisons among computation models
3. Propose the Harp framework with the MapCollective programming model to converge the HPC and Big Data domains
4. Implement the Harp framework with optimized collective communication operations as a plug-in to the Hadoop system
5. Provide guidelines for developing parallel machine learning algorithms on top of Harp
6. Implement two algorithms on Harp, K-means Clustering and SMACOF for MDS, using the "Allreduce" computation model with high speedup
7. Design a model rotation-based solution on Harp to implement three algorithms, CGS for LDA, SGD and CCD for MF, with fast model convergence speed and high scalability

Early work related to this thesis have appeared in seven publications, including information on large scale image clustering with optimizations on collective communication operations [16, 17], initial Harp design and implementation as a collective communication layer in the big data software stack [11, 18], analysis of model synchronization patterns in the LDA application [19], computation models [20], and the model rotation-based solution [21].

In the following sections, machine learning algorithms and their parallel computation models are first introduced. Then the research methodologies are described to show the convergence between HPC and big data techniques. Next, the programming interfaces, design, and implementation of the Harp framework are presented. Finally, several machine learning algorithm example implementations are presented, either using the "Allreduce" computation model or using the "Rotation" computation model. The conclusion is given at the end of the thesis.

[4] `https://portal.futuresystems.org`

2 Machine Learning Algorithms and Computation Models

This section is based on published papers [19, 20]. In this section, first the characteristics of machine learning algorithms and their parallelization are described, and then the categorization of computation models is given. Finally, the LDA application is used as an example to analyze the difference between computation models.

2.1 Machine Learning Algorithms

Iterative machine learning algorithms can be formulated as

$$A^t = F(D, A^{t-1}) \tag{1}$$

In this equation, D is the observed dataset, A are the model parameters to learn, and F is the model update function. The algorithm keeps updating model A until convergence (by reaching a stop criterion or a fixed number of iterations).

Parallelization can be performed by utilizing either the parallelism inside different components of the model update function F or the parallelism among multiple invocations of F. In the first form, the difficulty of parallelization lies in the computation dependencies inside F, which are either between the data and the model, or among the model parameters. If F is in a "summation form", such algorithms can be parallelized through the first category [6]. However, in large-scale machine learning applications, the algorithms picking random examples in model update perform asymptotically better than the algorithms in the summation form [22]. These algorithms are parallelized through the second form of parallelism. In the second category, the difficulty of parallelization lies in the dependencies between iterative updates of a model parameter. No matter which form of parallelization is used, when the dataset is partitioned into P parts, the parallel model update process can be generalized with only using one part of the data entries D_p as

$$A^t = F(D_p, A^{t-1}) \tag{2}$$

Obtaining the exact A^{t-1} is not feasible in parallelization. However, there are certain algorithm features which can maintain algorithm correctness and improve parallel performance.

I. ***The algorithms can converge even when the consistency of a model is not guaranteed to some extent.*** Algorithms can work on model A with an older version i when i is within bounds [23], as shown in

$$A^t = F(D_p, A^{t-i}) \tag{3}$$

By using a different version of A, Feature I breaks the dependency across iterations.

II. ***The update order of the model parameters is exchangeable.*** Although different update orders can lead to different convergence rates, they normally do not make the algorithm diverge. In the second form of parallelization, if F only accesses and updates one of the disjointed parts of the model parameters ($A_{p'}$), there is a chance of finding an arrangement on the order of model updates that allows independent model parameters to be processed in parallel while keeping the dependencies.

$$A^t_{p'} = F(D_p, A^{t-1}_{p'}) \tag{4}$$

2.2 Computation Model Survey

Since the key to parallelize machine learning algorithms is to parallelize the model update function, parallelization is not only viewed as training data-centric processing but also model parameter-centric processing. Based on model synchronization patterns and how the model parameters are used in parallel computation, parallel machine learning algorithms can be categorized into computation models. Through the understanding of model synchronization mechanisms, computation models are aimed to answer the following questions:

- What model parameters needs to be updated?
- When should the model update happen?
- Where should the model update occur?
- How is the model update performed?

Before the description of computation models, four attributes are introduced. These elements are key factors to computation models:

Worker In a computation model, each parallel unit is called a "worker." There are two levels of parallelism. In a distributed environment, each worker is a process, and the workers are synchronized through network communication. In a multi-thread environment, the workers are threads

which are coordinated through various mechanisms such as monitor synchronization.

Model Model parameters are the output of machine learning algorithms. Some algorithms may have multiple model parts. In these cases, the parallelization solution can store some model parts along with the training data and leaves the rest to synchronization.

Synchronized/Asynchronous Algorithm Computation models can be divided into those with synchronized algorithms and others with asynchronous algorithms. In synchronized algorithms, the computation progress on one worker depends on the progress on other workers; asynchronous algorithms lack this dependency.

The Latest/Stale Model Parameters Computation models use either the latest values or stale values from the model. The "latest" means that the current model used in computation is up-to-date and not modified simultaneously by other workers, while the "stale" indicates the values in the model are old. Since the computation model using the latest model maintains model consistency, its model output contains less approximation and is close to the output of the sequential algorithm.

These attributes derive four types of computation models, each of which uses a different means to handle the model and coordinate workers (see Figure 1). The computation model description focuses on the distributed environment. However, computation models can also be applied to a multi-thread environment. In a system with two levels of parallelism, model composition is commonly adapted with one type of computation model in the distributed environment and another in the multi-thread environment.

Computation Model A (Locking) This computation model uses a synchronized algorithm to coordinate parallel workers and guarantees each worker exclusive access to model parameters. Once a worker trains a data entry, it locks the related model parameters and prevents other workers from accessing them. When the related model parameters are updated, the worker unlocks the parameters. Thus the model parameters used in local computation is always the latest.

Computation Model B (Rotation) The next computation model also uses a synchronized algorithm. Each worker first takes a part of the shared model and performs training. Afterwards, the model is shifted between workers. During model rotation, each model parameter is updated by one worker at a time so that the model is consistent.

Computation Model C (Allreduce) This computation model applies a synchronized algorithm but with the stale model. In a single iteration, each worker first fetches all the model parameters

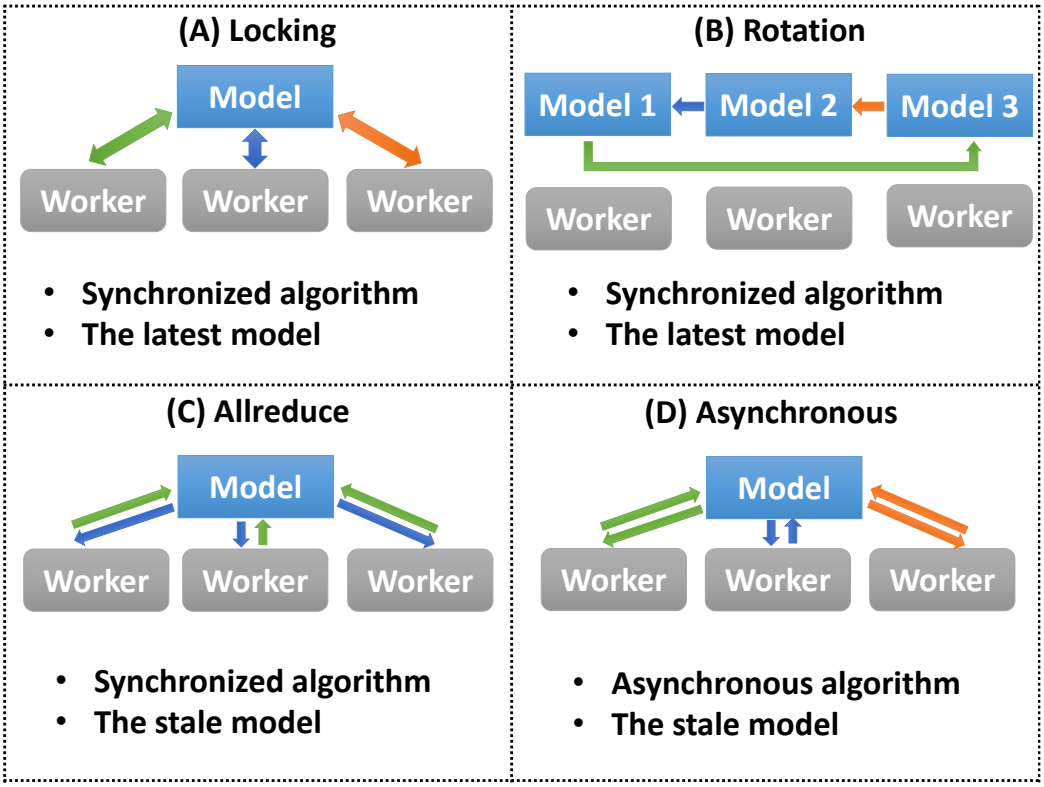


Figure 1: Computation Model Types

required by local computation. When local computation is completed, modifications of the local model from all workers are gathered to update the model.

Computation Model D (Asynchronous) With this computation model, an asynchronous algorithm employs the stale model. Each worker independently fetches related model parameters, performs local computation, and returns model modifications. Unlike the "Locking" computation model, workers are allowed to fetch or update the same model parameters in parallel. In contrast to "Rotation" and "Allreduce", there is no synchronization barrier.

Previous work shows many machine learning algorithms can be implemented in the MapReduce programming model [6]; later on, these algorithms are improved by in-memory computation and communication in iterative MapReduce [16, 17, 24]. However, the solution with broadcasting and reducing model parameters is a special case of the "Allreduce" computation model and is not immediately scalable when the model size is larger than the capacity of the local memory. As models may reach $10^{10} \sim 10^{12}$ parameters, Parameter Server type solutions [25, 26, 27] store the model on a set of server machines and use the "Asynchronous" computation model. Petuum [23], however, uses

Table 1: The Computation Models of LDA CGS Implementations

Implementation	Algorithm	Computation Model
PLDA [32]	CGS	Allreduce
PowerGraph LDA[a] [33]	CGS	Allreduce
Yahoo! LDA[b] [26, 27]	SparseLDA	Asynchronous
Peacock [34]	SparseLDA	Asynchronous & Rotation
Parameter Server [25]	CGS, etc.	Asynchronous
Petuum Bosen [23]	SparseLDA	Asynchronous
Petuum Strads[c] [28, 29, 30]	SparseLDA	Rotation & Asynchronous

[a] https://github.com/dato-code/PowerGraph/tree/master/toolkits/topic_modeling

[b] https://github.com/sudar/Yahoo_LDA

[c] https://github.com/petuum/bosen/wiki/Latent-Dirichlet-Allocation

a computation model mixed with both the "Allreduce" and "Asynchronous" computation models [23]. Petuum also implements the "Rotation" computation model [28, 29, 30].

In practice, it is difficult to decide which algorithm and computation model to use for the parallelization solution of a machine learning application. Taking the LDA application as an example, it can be solved by many different algorithms, such as CGS and Collapsed Variational Bayes (CVB) [3]. If CGS is selected, then the SparseLDA algorithm [31] is the most common implementation. Then comes another question: which computation model should be used to parallelize a machine learning algorithm? Table 1 lists various parallel CGS implementations, and it is possible to have other solutions that are not listed in Table 1, as long as the computation dependency is maintained. The difference between these solutions is discussed in the section below.

2.3 Latent Dirichlet Allocation

LDA is an important machine learning technique that has been widely used in areas such as text mining, advertising, recommender systems, network analysis and genetics. Though extensive research on this topic exists, the machine learning community is still endeavoring to scale it to web-scale corpora to explore more subtle semantics with a big model which may contain billions of model parameters [29]. The LDA application is commonly implemented using the SparseLDA algorithm. In this section, the model synchronization patterns and communication strategies of this algorithm are studied. The size of the model required for local computation is so large that sending such data to every worker results in communication bottlenecks. The required computation time is also great

due to the large model size. However, model size shrinks as the model converges. As a result, a faster model synchronization method can speed up model convergence, in which the model size shrinks and reduces the iteration execution time.

For two well-known LDA implementations, Yahoo! LDA uses the "Asynchronous" computation model while Petuum LDA uses the "Rotation" computation model. Though using different computation models, both solutions favor asynchronous communication methods. It not only avoids global waiting but also quickly makes the model update visible to other workers and thereby boosts model convergence. However, my research shows that model communication speed can be improved with collective communication methods. With collective communication optimization, the "Allreduce" computation model can outperform the "Asynchronous" computation model. Besides, in the "Rotation" computation model, using collective communication can further reduce model communication overhead compared with using asynchronous communication.

2.3.1 LDA Background

LDA modeling techniques can find latent structures inside the training data which are abstracted as a collection of documents, each with a bag of words. It models each document as a mixture of latent topics and each topic as a multinomial distribution over words. Then an inference algorithm works iteratively until it outputs the converged topic assignments for the training data. Similar to Singular Value Decomposition, LDA can be viewed as a sparse matrix decomposition technique on a word-document matrix, but it is rooted on a probabilistic foundation and has different computation characteristics.

Among the inference algorithms for LDA, CGS shows high scalability in parallelization [34, 35], especially compared with another commonly used algorithm, CVB (used in Mahout LDA[5]). CGS is a Markov chain Monte Carlo type algorithm. In the "initialize" phase, each training data point, or token, is assigned to a random topic denoted as z_{ij}. Then it begins to reassign topics to each token $x_{ij} = w$ by sampling from a multinomial distribution of a conditional probability of z_{ij}:

$$p\left(z_{ij} = k \mid z^{\neg ij}, x, \alpha, \beta\right) \propto \frac{N_{wk}^{\neg ij} + \beta}{\sum_w N_{wk}^{\neg ij} + V\beta} \left(M_{kj}^{\neg ij} + \alpha\right) \tag{5}$$

[5] https://mahout.apache.org/users/clustering/latent-dirichlet-allocation.html

Table 2: Sequential Algorithm Pseudo Code of CGS for LDA

Input: training data X, the number of topics K, hyperparameters α, β
Output: topic assignment matrix Z, topic-document matrix M, word-topic matrix N

```
1:  Initialize M, N to zeros
2:  for document j ∈ [1, D] do
3:      for token position i in document j do
4:          Z_ij = k ~ Mult(1/K)
5:          M_kj += 1; N_wk += 1
6:      end for
7:  end for
8:  repeat
9:      for document j ∈ [1, D] do
10:         for token position i in document j do
11:             M_kj −= 1; N_wk −= 1
12:             Z_ij = k' ~ p(Z_ij = k|rest) // [a]
13:             M_k'j += 1; N_wk' += 1
14:         end for
15:     end for
16: until convergence
```

[a] Sample a new topic according to Equation 5 using SparseLDA

Here superscript $\neg ij$ means that the corresponding token is excluded. V is the vocabulary size. N_{wk} is the token count of word w assigned to topic k in K topics, and M_{kj} is the token count of topic k assigned in document j. The matrices Z_{ij}, N_{wk}, and M_{kj}, are the model. Hyperparameters α and β control the topic density in the final model output. The model gradually converges during the process of iterative sampling. This is the phase where the "burn-in" stage occurs and finally reaches the "stationary" stage.

The sampling performance is more memory-bound than CPU-bound. The computation itself is simple, but it relies on accessing two large sparse model matrices in the memory. In Table 2, sampling occurs by the column order of the word-document matrix, called "sample by document". Although M_{kj} is cached when sampling all the tokens in a document j, the memory access to N_{wk} is random since tokens are from different words. Symmetrically, sampling can occur by the row order, called "sample by word". In both cases, the computation time complexity is highly related to the size of the model matrices. SparseLDA is an optimized CGS sampling implementation mostly used in state-of-the-art LDA trainers. In Line 10 of Table 2, the conditional probability is usually computed for each k with a total of K times, but SparseLDA decreases the time complexity to the number of non-zero elements in one row of N_{wk} and in one column of M_{kj}, both of which are much

smaller than K on average.

2.3.2 Big Model Problem

Sampling on Z_{ij} in CGS is a strict sequential procedure, although it can be parallelized without much loss in accuracy [35]. Parallel LDA can be performed in a distributed environment or a shared-memory environment. Due to the huge volume of training documents, the distributed environment is required for parallel processing. In a distributed environment, a number of compute nodes deployed with a single worker process apiece. Every process takes a partition of the training document set and performs the sampling procedure with multiple threads.

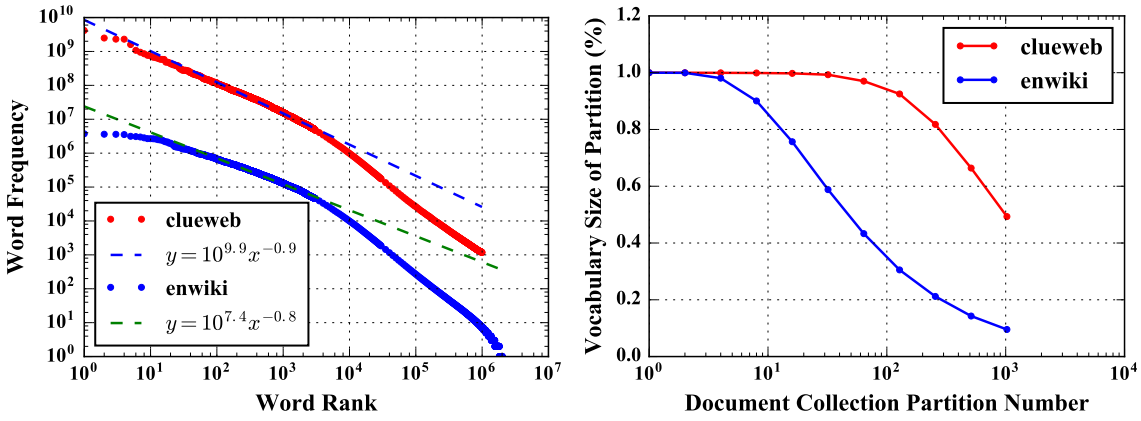
The LDA model contains four parts: I. Z_{ij} - topic assignments on tokens, II. N_{wk} - token counts of words on topics (word-topic matrix), III. M_{kj} - token counts of documents on topics (topic-document matrix), and IV. $\sum_w N_{wk}$ - token counts of topics. Here Z_{ij} is always stored along with the training tokens. For the other three, because the training tokens are partitioned by document, M_{kj} is stored locally while N_{wk} and $\sum_w N_{wk}$ are shared. For the shared model parts, a parallel LDA implementation may use the latest model or the stale model in the sampling procedure.

Now it is time to calculate the size of N_{wk}, a huge but sparse $V \times K$ matrix. The word distribution in the training data generally follows a power law. If the words based on their frequencies is sorted from high to low, for a word with rank i, its frequency is $freq(i) = C \cdot i^{-\lambda}$. Then for W, the total number of training tokens, there is

$$W = \sum_{i=1}^{V} freq(i) = \sum_{i=1}^{V} (C \cdot i^{-\lambda}) \approx C \cdot (lnV + \gamma + \frac{1}{2V}) \tag{6}$$

To simplify the analysis, λ is considered as 1. Since C is a constant equal to $freq(1)$, then W is the partial sum of the harmonic series which have logarithmic growth, where γ is the Euler-Mascheroni constant ≈ 0.57721. The real model size S depends on the count of non-zero cells. In the "initialize" phase of CGS, with random topic assignment, a word i gets $max(K, freq(i))$ non-zero elements. If $freq(J) = K$, then $J = C/K$, and there is:

$$S_{init} = \sum_{i=1}^{J} K + \sum_{i=J+1}^{V} freq(i) = W - \sum_{i=1}^{J} freq(i) + \sum_{i=1}^{J} K = C \cdot (lnV + lnK - lnC + 1) \tag{7}$$



(a) Zipf's Law of the Word Frequencies (b) Vocabulary Size vs. Document Partitioning

Figure 2: Words' Distribution on the Training Documents

The initial model size S_{init} is logarithmic to matrix size $V \cdot K$, which means $S << V \cdot K$. However, this does not mean S_{init} is small. Since C can be very large, even $C \cdot ln(VK)$ can result in a large number. In the progress of iterations, the model size shrinks as the model converges. When a stationary state is reached, the average number of topics per word drops to a certain small constant ratio of K, which is determined by the concentration parameters α, β and the nature of the training data itself.

The local vocabulary size V' on each process determines the model volume required for local computation. When documents are randomly partitioned to N processes, every word with a frequency higher than N has a high probability of occurring on all the processes. If $freq(L) = N$ at rank L, then $L = \frac{W}{(lnV+\gamma)\cdot N}$. For a large training dataset, the ratio between L and V is often very high, indicating that local computation requires most of the model parameters. Figure 2 shows the difficulty of controlling local vocabulary size in random document partitioning. When 10 times more partitions are introduced, there is only a sub-linear decrease in the vocabulary size per partition. The "clueweb" and "enwiki" datasets are used as examples (see Section 2.3.5). In "clueweb", each partition gets 92.5% of V when the training documents are randomly split into 128 partitions. "enwiki" is around 12 times smaller than "clueweb". It gets 90% of V with 8 partitions, keeping a similar ratio. In summary, though the local model size reduces as the number of compute nodes grows, it is still a high percentage of V in many situations.

In conclusion, there are three key properties of the LDA model:

1. The model parameters required in local computation is a high percentage of all the model

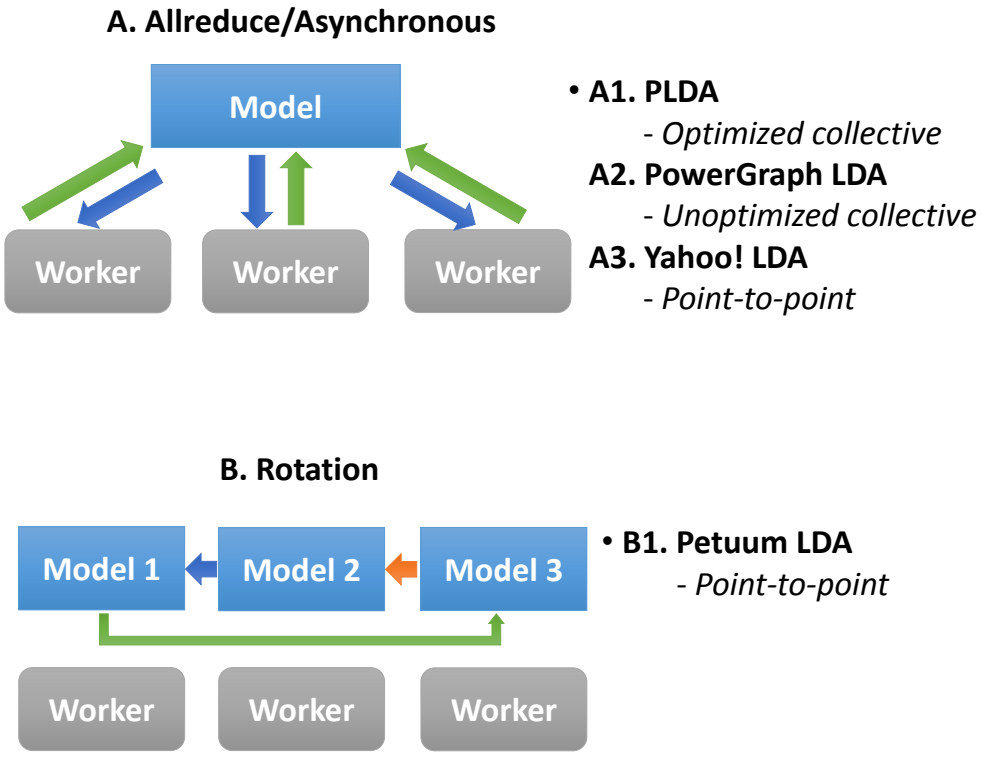


Figure 3: Synchronization Methods under Different Computation Models

parameters.

2. The local computation time is related to the local model size.
3. The initial model size is huge but it reduces as the model converges.

These properties indicate that the communication optimization in model synchronization is necessary because it can accelerate the model update process and result in a huge benefit in computation and communication of later iterations.

Of the various synchronization methods used in state-of-the-art implementations, they can be categorized into two types (see Figure 3). In the first type, the parallelization either uses the "Allreduce" computation model or the "Asynchronous" computation model. Both computation models work on stale model parameters. In PLDA, without storing a shared model, it synchronizes local model parameters through a MPI "allreduce" operation [36]. This operation is routing optimized, but it does not consider the model requirement in local computation, causing high memory usage and high communication load. In PowerGraph LDA, model parameters are fetched and returned directly in a synchronized way. Though it communicates less model parameters compared with PLDA, the

performance is low for lack of routing optimization. Unlike the two implementations above which use the "Allreduce" computation model, a more popular implementation, Yahoo! LDA, follows the "Asynchronous" computation model. Yahoo! LDA can ease the communication overhead with asynchronous point-to-point communication, however, its model update rate is not guaranteed. A word's model parameters may be updated either by changes from all the training tokens, a part of them, or even no change. A solution to this problem is to combine the "Allreduce" computation model and the "Asynchronous" computation model. This is implemented in Petuum Bosen LDA [23]. In the second type, the "Rotation" computation model is used. Currently, only Petuum LDA is in this category. In its implementation, parameters are sent to the neighbor asynchronously.

A better solution for the first type of synchronization method can be a conjunction of PLDA and PowerGraph LDA with new collective communication optimizations which include both routing optimization and reduced data size for communication. There are three advantages to such a strategy. First, considering the model requirement in local computation, it reduces the model parameters communicated. Second, it optimizes routing through searching "one-to-all" communication patterns. Finally, it maintains the model update rate compared with the asynchronous method. For the second type of synchronization method, using collective communication is also helpful because it maximizes bandwidth usage between compute nodes and avoids flooding the network with small messages, which is what Petuum LDA does.

2.3.3 Model Synchronization Optimizations

New solutions with optimized collective communication operations to parallelize the SparseLDA algorithm are developed. Model parameters are distributed among processes. Two model synchronization methods are used. In the first method, a set of local models are defined on each process. Each local model is considered a local version of the global model. The synchronization has two steps. The first step redistributes the model parameters related to the local computation from the global model to local models. The second step reduces the updates from different local models to one in the global model. Model parameters are packed into chunks and sent to avoid small message flooding. Routing optimized broadcasting [36] is used if "one-to-all" communication patterns are detected on a set of model parameters. In the second method, "rotate" considers processes in a ring topology and shifts the model chunks from one process to the next neighbor. The model parameters

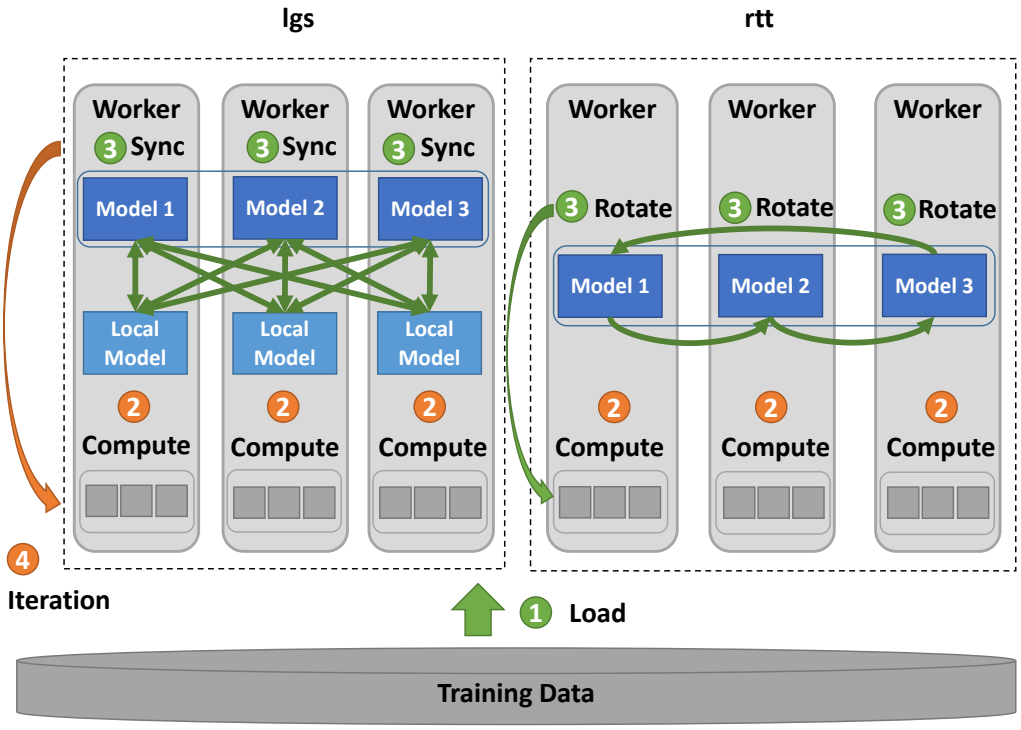


Figure 4: LDA implementations with Optimized Communication

are partitioned based on the range of word frequencies in the training dataset. The lower the frequency of the word, the higher the word ID given. Then the word IDs are mapped to process IDs based on the modulo operation. In this way, each process contains model parameters with words whose frequencies are ranked from high to low. In the first synchronization method, this kind of model partitioning can balance the communication load. In the second synchronization method, it can balance the computation load on different processes between two times of model shifting.

As a result, two parallel LDA implementations are presented (see Figure 4). One is "lgs" (an abbreviation of "local-global synchronization"), and another is "rtt" (an abbreviation of "rotate"). In both implementations, the computation and communication are pipelined to reduce the synchronization overhead, i.e., the model parameters are sliced in two and communicated in turns. Model Part IV is synchronized through an "allreduce" operation at the end of every iteration. In the local computation, both "lgs" and "rtt" use the "Asynchronous" computation model. However, "lgs" samples by document while "rtt" samples by word. All these are done to keep the consistency between implementations for unbiased communication performance comparisons in experiments. Of

Table 3: Training Data and Model Settings in the Experiments

Dataset	Number of Docs	Number of Tokens	Vocabulary	Doc Length Mean/SD	Number of Topics	Initial Model Size
clueweb	50.5M	12.4B	1M	224/352	10K	14.7GB
enwiki	3.8M	1.1B	1M	293/523	10K	2.0GB
bi-gram	3.9M	1.7B	20M	434/776	500	5.9GB

course, for "rtt", since each model shifting only gives a part of the model parameters for the local computation, sampling by word can easily reduce the time used for searching tokens which can be sampled.

2.3.4 Experiments

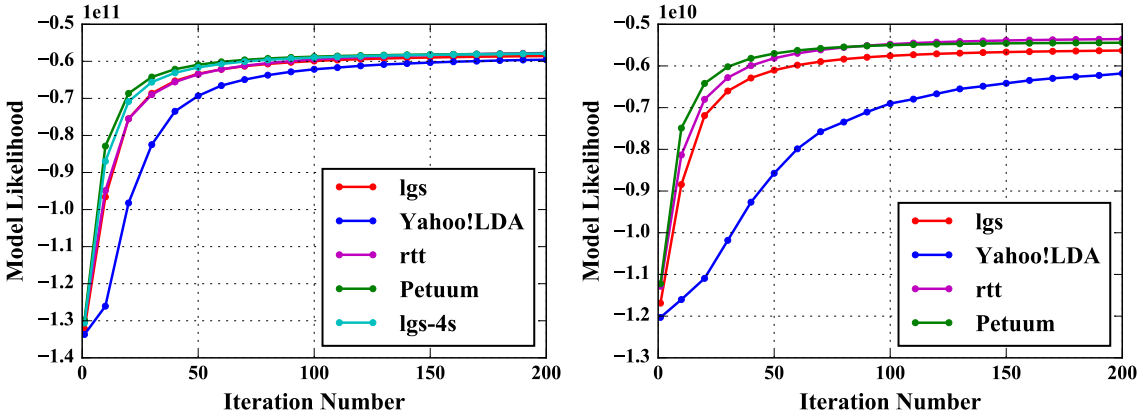
Experiments are done on an Intel Haswell cluster. This cluster contains 32 nodes each with two 18-core 36-thread Xeon E5-2699 processors and 96 nodes each with two 12-core 24-thread Xeon E5-2670 processors. All the nodes have 128GB memory and are connected with 1Gbps Ethernet (eth) and Infiniband (ib). For testing, 31 nodes with Xeon E5-2699 and 69 nodes with Xeon E5-2670 are used to form a cluster of 100 nodes, each with 40 threads. All the tests are done with Infiniband through IPoIB support.

"clueweb"[6], "enwiki", and "bi-gram"[7] are three datasets used (see Table 3). The model settings are comparable to other research work [29], each with a total of 10 billion parameters. α and β are both fixed to a commonly used value of 0.01 to exclude dynamic tuning. Several implementations are tested: "lgs", "lgs-4s" ("lgs" with 4 rounds of model synchronization per iteration, each round with 1/4 of the training tokens), and "rtt". To evaluate the quality of the model outputs, the model likelihood on the words' model parameters is used to monitor model convergence. These LDA implementations are compared with Yahoo! LDA and Petuum LDA, and thereby how communication methods affect LDA performance are learned by studying the model convergence speed.

The model convergence speed is first measured by analyzing model outputs on Iteration 1, 10, 20... 200. In an iteration, every training token is sampled once. Thus the number of model updates in each iteration is equal. Then how the model converges with the same amount of model updates is measured. On "clueweb" (see Figure 5(a)), Petuum has the highest model likelihood on all

[6] 10% of ClueWeb09, a collection of English web pages, `http://lemurproject.org/clueweb09.php/`

[7] Both "enwiki" and "bi-gram" are English articles from Wikipedia, `https://www.wikipedia.org`

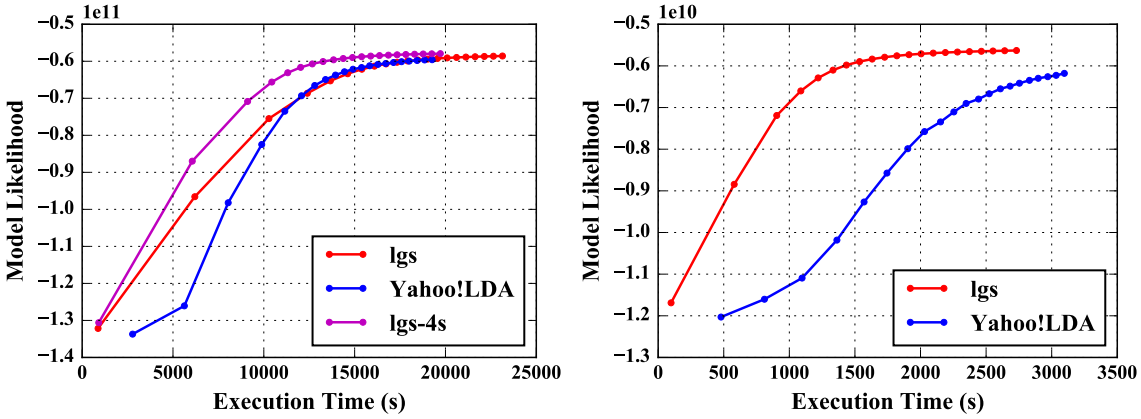


(a) Model Convergence Speed on "clueweb" (b) Model Convergence Speed on "enwiki"

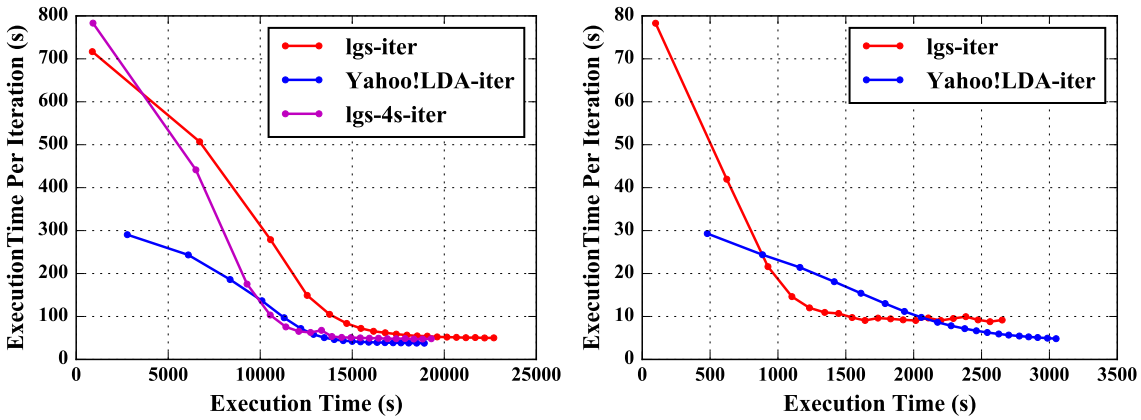
Figure 5: Model Convergence Speed by Iteration Count

iterations. Due to the preference of "rtt" in using stale thread-local model parameters in multi-thread sampling, the convergence speed is slower. The lines of "rtt" and "lgs" are overlapped indicating their similar convergence speeds. In contrast to "lgs", the convergence speed of "lgs-4s" is as high as Petuum. This shows that increasing the number of model update rounds improves convergence speed. Yahoo! LDA has the slowest convergence speed because asynchronous communication does not guarantee all model updates are seen in each iteration. On "enwiki" (see Figure 5(b)), as before, Petuum achieves the highest accuracy out of all iterations. "rtt" converges to the same model likelihood level as Petuum at iteration 200. "lgs" demonstrates slower convergence speed but still achieves high model likelihood, while Yahoo! LDA has both the slowest convergence speed and the lowest model likelihood at iteration 200. Though the number of model updates is the same, an implementation using the stale model converges slower than one using the latest model. For those using the stale model, "lgs-4s" is faster than "lgs" while "lgs" is faster than Yahoo! LDA. This means by increasing the number of model synchronization rounds, the model parameters used in computation are newer, and the convergence speed is improved.

Then the model convergence speed is measured by the execution time. First the execution speed between "lgs" and Yahoo! LDA is compared. On "clueweb", the convergence speed is shown based on the elapsed execution time (see Figure 6(a)). Yahoo! LDA takes more time to finish Iteration 1 due to its slow model initialization, which demonstrates that it has a sizable overhead on the communication end. In later iterations, while "lgs" converges faster, Yahoo! LDA catches up after 30 iterations. This observation can be explained by the slower computation speed of the current Java



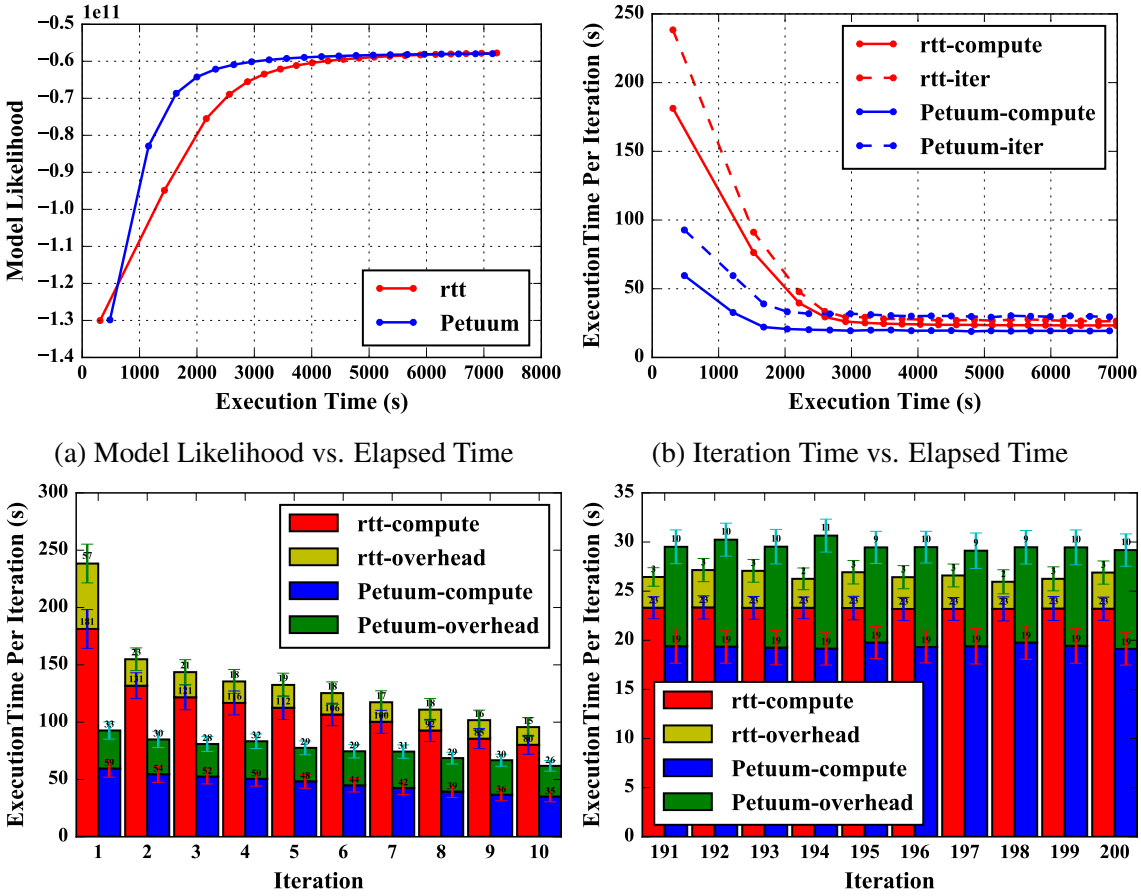
(a) Model Convergence Speed on "clueweb" (b) Model Convergence Speed on "enwiki"



(c) Per Iteration Time on "clueweb" (d) Per Iteration Time on "enwiki"

Figure 6: Comparison between "lgs" and Yahoo! LDA

implementation. To counteract the computation overhead, the number of model synchronization rounds per iteration is increased to four. Thus the computation overhead is reduced by using a newer and smaller model. Although the execution time of "lgs-4s" is still slightly longer than Yahoo! LDA, it obtains higher model likelihood and maintains faster convergence speed during the whole execution. Similar results are shown on "enwiki", but this time "lgs" not only achieves higher model likelihood but also has faster model convergence speed throughout the whole execution (see Figure 6(b)). From both experiments, it is shown that though the computation is slow in "lgs", with collective communication optimization, the model size quickly shrinks so that its computation time is reduced significantly. At the same time, although Yahoo! LDA does not have any extra overhead other than computation in each iteration, its iteration execution time reduces slowly because it keeps computing with an older model (see Figure 6(c)(d)).



(a) Model Likelihood vs. Elapsed Time

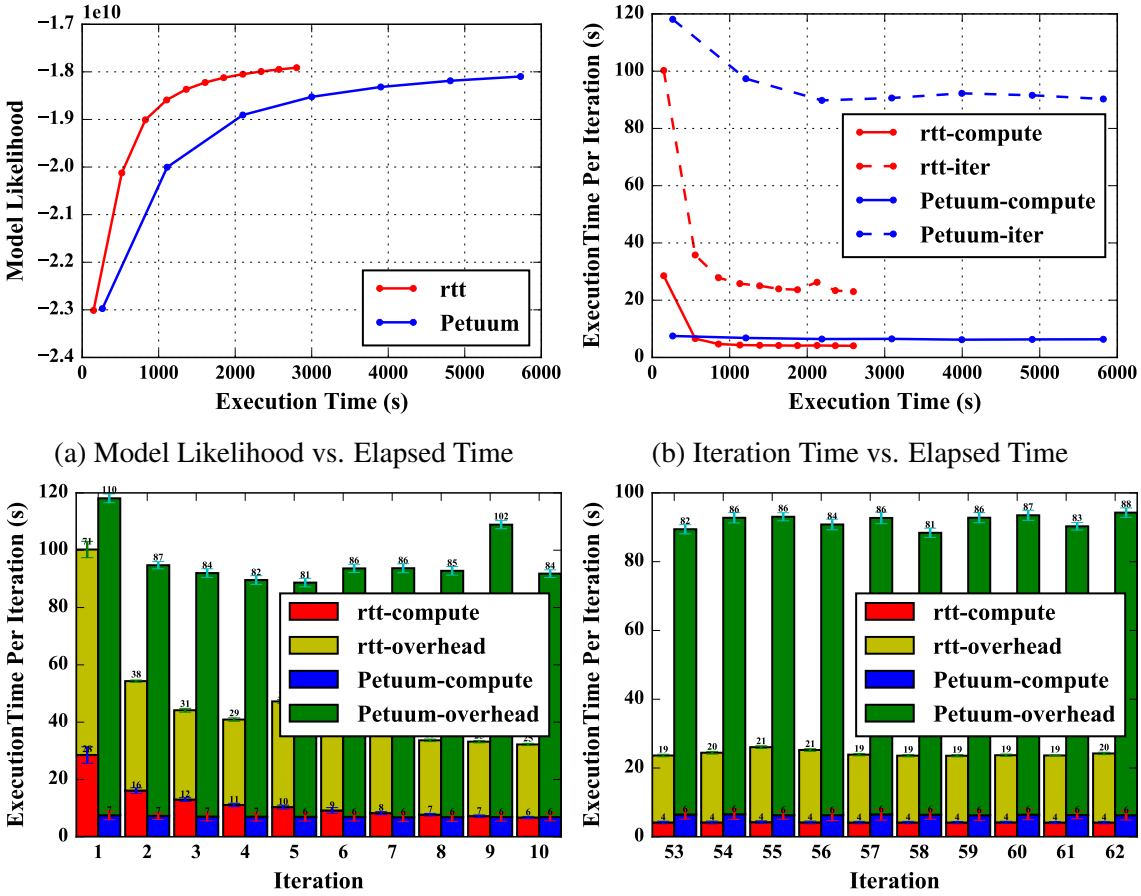
(b) Iteration Time vs. Elapsed Time

(c) First 10 Iteration

(d) Final 10 Iteration

Figure 7: Comparison between "rtt" and Petuum on "clueweb"

Next, "rtt" and Petuum LDA are compared on "clueweb"and "bi-gram". On "clueweb", the execution times and model likelihood achieved on both sides are similar (see Figure 7(a)). Both are around 2.7 times faster than the results in "lgs" and Yahoo! LDA. This is because they use the latest model parameters for sampling, which quickly reduces the model size for further computation. Besides, sampling by word leads to better local computation performance compared with sampling by document due to less model parameter fetching/updating conflict in the "Asynchronous" computation model. Though "rtt" has higher computation time compared with Petuum LDA, the communication overhead per iteration is lower. When the execution arrives at the final few iterations, while computation time per iteration in "rtt" is higher, the whole execution time per iteration becomes lower (see Figure 7(b)(c)(d)). This is because Petuum communicates each word's model parameters in small messages and generates high overhead. On "bi-gram", the results show that



(a) Model Likelihood vs. Elapsed Time

(b) Iteration Time vs. Elapsed Time

(c) First 10 Iteration

(d) Final 10 Iteration

Figure 8: Comparison between "rtt" and Petuum on "bi-gram"

Petuum does not perform well when the number of words in the model increases. The high overhead in communication causes the convergence speed to be slow, and Petuum cannot even continue executing after 60 iterations due to a memory outage (see Figure 8).

In sum, the model properties in parallel LDA computation suggest that using collective communication optimizations can improve the model update speed, which allows the model to converge faster. When the model converges quickly, its size shrinks greatly, and the iteration execution time also reduces. Optimized collective communication methods is observed to perform better than asynchronous methods. "lgs" results in faster model convergence and higher model likelihood at iteration 200 compared to Yahoo! LDA. On "bi-gram", "rtt" shows significantly lower communication overhead than Petuum LDA, and the total execution time of "rtt" is 3.9 times faster. On "clueweb", although the computation speed of the first iteration is 2- to 3-fold slower for "rtt" compared to

Petuum LDA, the total execution time remains similar.

Despite the implementation differences between "rtt", "lgs", Yahoo! LDA, and Petuum LDA, the advantages of collective communication methods are evident. Compared with asynchronous communication methods, collective communication methods can optimize routing between parallel workers and maximize bandwidth utilization. Though collective communication will result in global waiting, the resulting overhead is not as high as speculated when the load-balance is handled. The chain reaction set off by improving the LDA model update speed amplifies the benefits of using collective communication methods. When putting all the performance results together, it is also clear that both "rtt" and Petuum are remarkably faster than the remaining implementations. This shows in LDA parallelization, using the "Rotation" computation model can achieve higher performance compared with the "Allreduce" and "Asynchronous" computation models. Between the "Allreduce" and "Asynchronous" computation models, "lgs" proves to be faster than Yahoo! LDA at the beginning, but at later stages, their model convergence speed tends to overlap. Through adjusting the number of model synchronization frequencies to 4 per iteration, "lgs-4s" exceeds Yahoo! LDA from start to finish. This means with the optimized collective communication, the "Allreduce" computation model can exceed the "Asynchronous" computation model. From these results, it can be concluded that the selection of computation models, combined with the details of computation load-balancing and communication optimization, needs to be carefully considered in the implementation of a parallel machine learning algorithm, as it is key to execution performance.

3 Solving the Big Data Problem with HPC Methods

The LDA application presents an example which shows that selecting a proper computation model is important to machine learning algorithm parallelization, and using collective communication optimization can improve model synchronization speed. However, existing tools for parallelization have limited support to computation models and collective communication techniques. Since collective communication has been commonly used in the HPC domain, it is a chance to adapt it to big data machine learning and derive an innovative solution.

3.1 Related Work

Before the emergence of big data tools, MPI is the primary tool used for parallelization in the HPC domain. Other than "send" and "receive" calls, MPI provides collective communication operations such as "broadcast", "reduce", "allgather", and "allreduce". These operations provide efficient communication performance through optimized routing. However, these operations only describe the relations between parallel processes in collective communication. When fine-grained model synchronization is required, these collective communication operations lack a mechanism to identify the relations between model updates and parameters in the synchronization. In this case, lots of "send" and "receive" calls are used, which makes the application code complicated.

In contrast with MPI, big data tools focus on synchronization based on the relations between communication data entries. In MapReduce [5], input data are read as Key-Value pairs, and in the "shuffle" phase, intermediate data are regrouped based on keys. So the communication pattern depends on the distribution of intermediate data. Initially, MapReduce proved very successful as a general tool to process many problems but was later considered not optimized for many important analytics, especially those machine learning algorithms involving iterations. The rationale is that the MapReduce frameworks have to repeatedly load training data from distributed file systems (HDFS) for each iteration. Iterative MapReduce frameworks such as Twister [7] and Spark [8] improve the performance through caching invariable training data. In addition, Pregel [9], Giraph[8], and GraphLab [33, 37] abstract the training data as a graph, cache, and process it in iterations. Synchronization is performed as messaging along the edges between neighbor vertices. Though the

[8] `https://giraph.apache.org`

synchronization relations between data entries/model parameters are expressed in these tools, there are limitations. In MapReduce, the dependency between model parameters and the local training data is not well solved. Model parameters are only broadcasted to the workers or reduced to one in the execution flow, making the implementation hard to scale. This is seen in the K-means Clustering implementations of Mahout on Hadoop[9] or Spark MLlib[10]. Other applications' complicated synchronization dependency also require multiple ways of model synchronization (e.g. the SparseLDA algorithm). However, both MapReduce and Graph tools follow the "Allreduce" computation model. It is impossible for developers to parallelize SparseLDA with the "Rotation" computation model within these frameworks.

Routing optimization is another important feature which is missing in existing big data tools. For example, in K-means with Lloyd's algorithm [12], the training data (high dimensional points) can be easily split and distributed to all the workers, but the model (centroids) have to be synchronized and redistributed to all the parallel workers in successive iterations. Mahout on Hadoop chooses to reduce model updates from all the Map tasks in one Reduce task, generate the new model, store it on HDFS, and let all the Map tasks read the model back to memory. The whole process can be summarized as "reduce-broadcast". According to C.-T. Chu et al. [6], this pattern can be applied to many other machine learning algorithms, including Logistic Regression, Neural Networks, Principal Component Analysis, Expectation Maximization, and Support Vector Machines, all of which follow the statistical query model with the summation form in their model update formulas. However, when both the size of the model and the number of workers grow large, this method becomes inefficient. In K-means Clustering, the time complexity of this communication process is $O(pkd)$, where p is the number of workers, k is the number of centroids, and d is the number of dimensions per centroid/point. In my initial research work, a large-scale K-means clustering on Twister is studied. Image features from a large collection of seven million social images, each representing a point in a high dimensional vector space, are clustered into one million clusters [16, 17]. This clustering application is split into five stages in each iteration: "broadcast", "map", "shuffle", "reduce", and "combine". By applying a three-stage synchronization of "regroup-gather-broadcast", the overhead of data synchronization can be reduced to $O(3kd)$. Furthermore, if "regroup-allgather" is applied

[9] `https://mahout.apache.org`

[10] `https://spark.apache.org/docs/latest/ml-guide.html`

Table 4: Programming Models and Synchronization Patterns of Big Data Tools

Tool	Programming Model	Synchronization Pattern
MPI	a set of parallel workers are spawned with communication support between them	send/receive or collective communication operations
Hadoop	(iterative) MapReduce, DAG-like job execution flow may be supported	disk-based shuffle between the Map stage and the Reduce stage
Twister		in-memory regroup between the Map stage and the Reduce stage; "broadcast" and "aggregate"
Spark		RDD transformations on RDD; "broadcast" and "aggregate"
Giraph	BSP model, data are expressed as vertices and edges in a graph	graph-based message communication following the Pregel model (vertex-based partitioning, messages are sent between neighbor vertices)
Hama		graph-based communication following the Pregel model or direct message communication between workers
GraphLab (Turi)		graph-based communication through caching and fetching of ghost vertices and edges, or the communication between a master vertex and its replicas in the PowerGraph (GAS) model
GraphX		graph-based communication supports both the Pregel model and the PowerGraph model
Parameter Server	BSP model, or loosely synchronized on the parameter server	asynchronous "push" and "pull" calls are used for communicating model parameters between parameter servers and workers
Petuum		in addition to asynchronous "push" and "pull" calls, the framework allows scheduling model parameters between workers

directly, the communication time can even be reduced to $O(2kd)$. The LDA application is another example to show the advantages of collective communication with routing optimization. As what has been discussed in the previous section, both Parameter Server type Yahoo! LDA and Petuum are inefficient in communication due to asynchronous parameter-based point-to-point communication.

In sum, existing tools are listed according to their programming models and synchronization patterns (see Table 4). The first one is MPI. It spawns multiple parallel processes and performs synchronization through "send"/"receive" calls or collective communication operations based on the process relations. Next is the MapReduce type; MapReduce systems such as Hadoop describes the parallelization as processing inputs as Key-Value pairs in the Map tasks, generating intermediate Key-Value pairs, and then shuffling and reducing them. It became popular thanks to its sim-

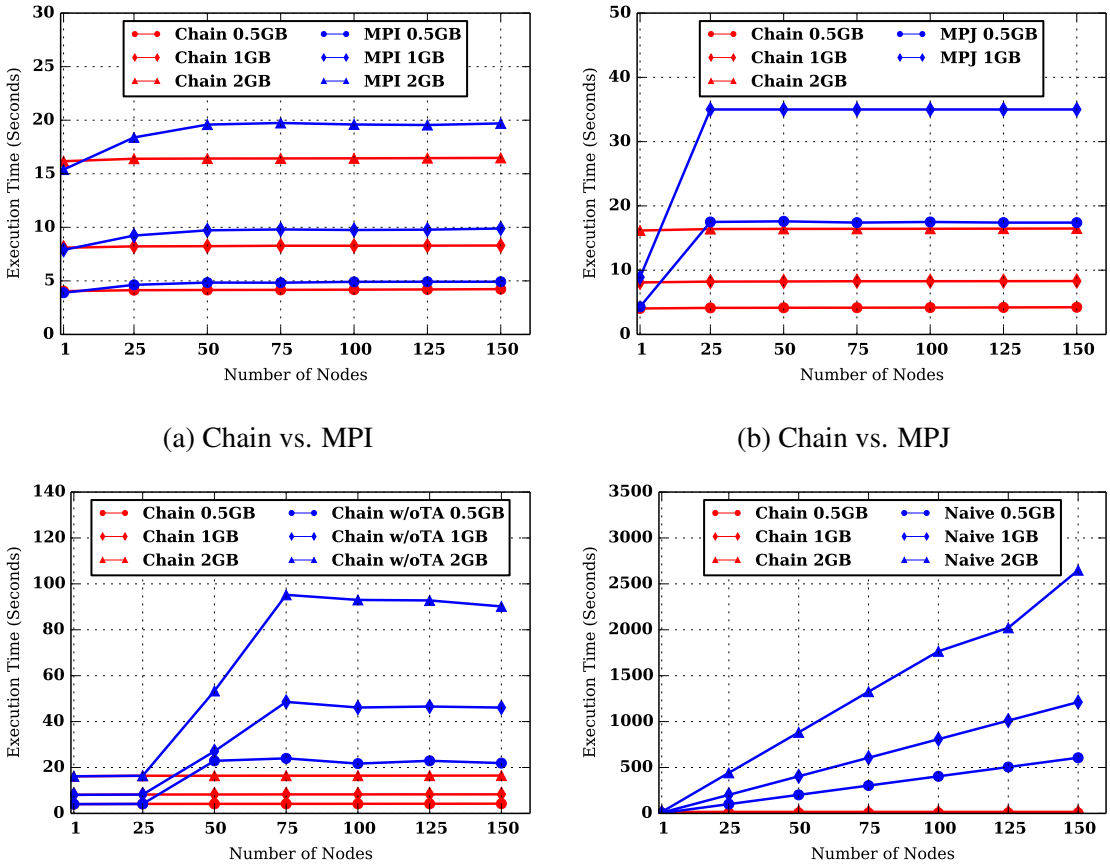
plicity, yet is still slow when running iterative algorithms. Frameworks like Twister, Spark, and HaLoop [38] solved this issue by caching the training data and extending MapReduce to iterative MapReduce. In Spark, multiple MapReduce jobs can form a directed acyclic execution graph with additional work flow management to support complicated data processing. The third type is the graph processing tools, which abstracts data as vertices and edges and executes in the BSP (Bulk Synchronous Parallel) model. Pregel and its open source version Giraph and Hama[11] follow this design. In contrast, GraphLab [37] (now called Turi[12]) abstracts data as a graph but uses consistency models to control vertex value updates (no explicit message communication calls). GraphLab was later enhanced with PowerGraph [33] abstractions to reduce the communication overhead. This was also used by GraphX [39]. The fourth type of tools directly serve machine learning algorithms by providing programming models for model parameter synchronization. These tools include Parameter Server and Petuum. Parameter Server does not force global synchronization. Parameters are stored in a separate group of servers. Parallel workers can exchange model updates with servers asynchronously. In addition to the model synchronization between workers and servers, Petuum also allows for scheduling and shifting model parameters between workers.

3.2 Research Methodologies

Based on the requirements of model synchronization in parallel machine learning algorithms and the discussion about the status quo of how the computation models and collective communication techniques are applied in existing tools, it is necessary to build a separate communication layer with high level programming interfaces to provide a rich set of communication operations, granting users flexibility and easiness to develop machine learning applications. There are three challenges derived from prior research:

Express Communication Patterns from Different Tools From the four computation models, the "Allreduce" and "Rotation" computation models can be expressed with collective communication operations. As what is shown in the related work, each tool has its own synchronization operations, either based on the worker relations or the data entry relations. For a unified collective communication layer, it is necessary to unite different synchronization patterns into one layer

[11] `https://hama.apache.org`
[12] `https://turi.com/`



(a) Chain vs. MPI

(b) Chain vs. MPJ

(c) Chain vs. Chain w/o topology-awareness

(d) Chain vs. the Naive Method

Figure 9: The "Chain Broadcast" Method

and provide a set of programming interfaces for parallel machine learning applications which can re-distribute training data entries and synchronize model parameters.

Optimize Collective Communication Operations The performance of collective communication varies significantly based on the implementation. The key to high performance is routing optimization. Through a specially designed routing order, the communication can maximize the bandwidth usage between workers. In the past, several works have tried to improve the performance of "broadcast" to quickly synchronize the model parameters [16, 17, 24, 40]. The results of my work [17] are presented in Figure 9. They show that when the bandwidth is limited (1 Gbps), topology-aware pipeline-based chain broadcasting can achieve similar performance compared with Open MPI 1.4.1 (see Figure 9(a)) and runs 4 times faster compared with MPJ 0.38 (see Figure 9(b)). When the size of the communication data becomes huge, the main communication overhead is de-

termined by the bandwidth. However, the time taken on broadcasting does not increase much when the number of nodes increases. As a result, for each collective communication operation designed, it is important to provide routing-optimized implementation based on the size and the distribution of the communication and the communication patterns among these data.

Apply Collective Communication Operations to Machine Learning Applications Besides designing and implementing collective communication operations, investigating the computation and communication characteristics of each machine learning application is necessary in order to find the proper computation model with balanced computation load and optimized collective communication operations. These factors are important to both algorithm convergence and execution performance. In this thesis, with several parallel machine learning algorithm examples, guidelines are provided for selecting computation models, choosing collective communication operations, and balancing computation load.

4 Harp Programming Model

Based on Sections 2 and 3, I have identified the need for a new framework that is able to meet the various needs of machine learning algorithm parallelization and converge the technologies from both the HPC and Big Data domains. In this section, I introduce Harp, a new framework that can program machine learning algorithms with the MapCollective programming model. The description covers the basic execution flow, the related data, and the communication abstractions.

4.1 MapCollective Programming Model

The MapCollective programming model is derived from the original MapReduce programming model. For the input data, similar to MapReduce, they are read from HDFS as Key-Value pairs to Map tasks. However, instead of using Map/Reduce tasks and the shuffling procedure to exchange Key-Value pairs, the MapCollective programming model keeps the Map tasks alive and allows data exchange through collective communication operations (see Figure 10). Therefore, this programming model follows the BSP model and enables two levels of parallelism. At the inter-node level, each worker is a Map task where the collective communication operations happen. The second is the intra-node level for multi-thread processing inside Map tasks. Thread-level parallelism is not mandatory in the MapCollective programming model, but it can maximize memory sharing and parallelism inside one machine. The fault tolerance in this programming model poses a challenge because its execution flow becomes very flexible when the collective communication operations are invoked. Currently, job-level failure recovery is applied. An application with a large number of iterations can be separated into several jobs, each of which containing multiple iterations. This naturally forms check points between iterations. Simultaneously, worker-level recovery by resynchronizing execution states between new launched workers and other old live workers is also under investigation.

4.2 Hierarchical Data Interfaces

Key-Value pairs are still used as interfaces for processing input data, but to support various collective communication patterns, data types are abstracted in a hierarchy. Data in collective communication are horizontally abstracted as primitive arrays or objects and constructed into partitions and tables

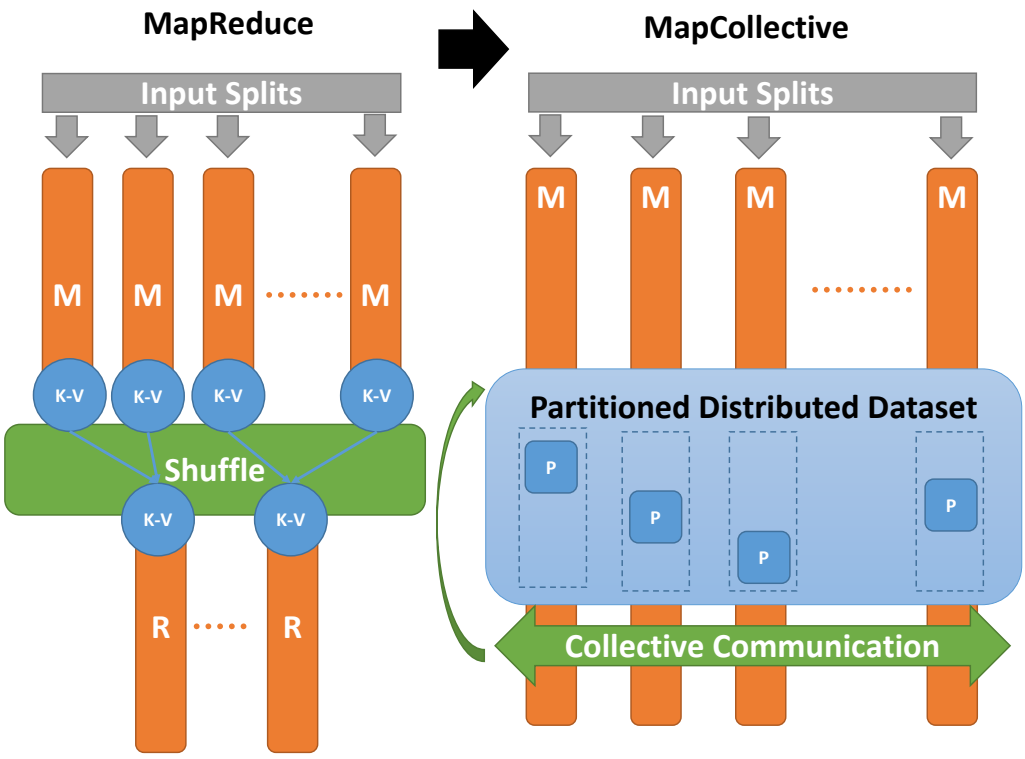


Figure 10: MapReduce vs. MapCollective

(see Figures 11 and 12). Any type which can be sent or received is under the interface "Transferable". Based on the component type of an array, there can be byte array, short array, int array, float array, long array, or double array. Users can also define an arbitrary object under the "Writable" interface with code which tells the framework how to serialize or deserialize the object. Both arrays and objects are under the "Simple" interface. Another type under the interface "Transferable" is "Partition". It contains a partition ID and a partition body, which is either an array or a "Writable" object. The partition ID is used for indexing the partition body during the procedure of collective communication. A collection of partitions inside each Map task are held by a "Table". In a collective communication operation, tables in different Map tasks are associated with each other to describe a distributed dataset. Partition IDs are used to direct partition re-distribution or consolidation in the tables. Since each table is also equipped with a combiner, if a partition received uses the same ID as the partition in the table, it will solve the ID conflict by combining them into one. The partition-based data abstraction provides a coarse-grained mapping between the communication data. Thus the collective communication operations based on data relations can be performed. In addition to

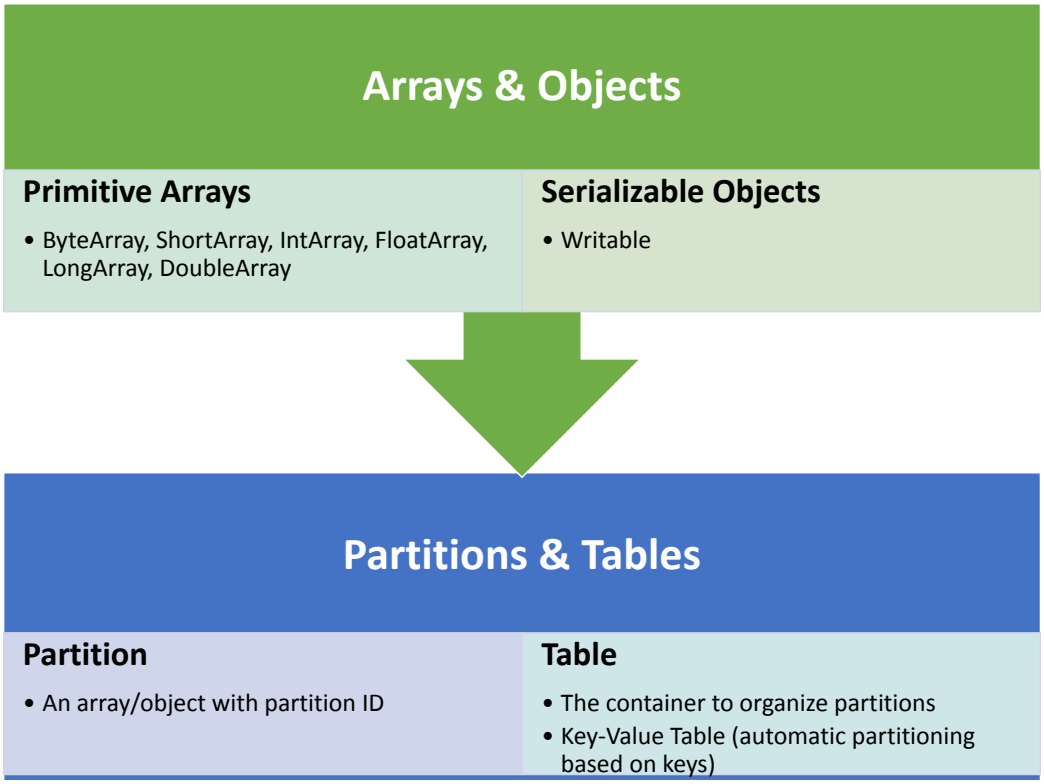


Figure 11: Data Interface Hierarchy

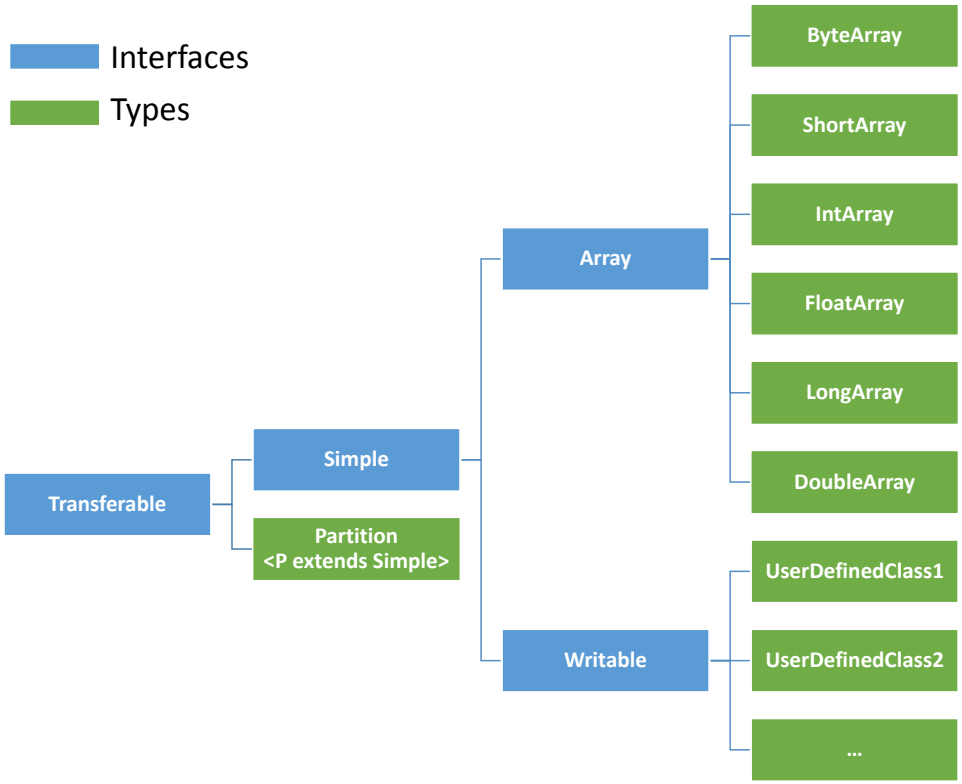


Figure 12: Data Interface Inheritance of “Array” and “Writable”

partition ID-based indexing, a type of "Key-Value Table" is still provided if users needs customized indexing for the value objects. Key-Value tables allow users to manage Key-Value pairs through providing auto partitioning based on keys. In this way, Key-Value pairs are organized as Key-Value partitions in Key-Value tables, and they are redistributed based on Key-Value partitions in collective communication operations.

4.3 Collective Communication Operations

Collective communication is operated on top of one or more set of user-defined tables on all the workers. Currently, four types of communication operations are observed in existing tools:

1. MPI collective communication operations [36]:
   e.g. "broadcast", "reduce", "allgather", and "allreduce"
2. MapReduce "shuffle-reduce" operation:
   e.g. regroup operation with combine
3. Graph communication:
   e.g. edge communication in Pregel, vertex communication in the GAS programing model
4. Operations from machine learning frameworks such as Parameter Server and Petuum:
   e.g. "push" and "pull" model parameters from the the global model, or rotate global model parameters between workers

Thus in Harp, seven collective communication operations are defined based on the observations above (see Table 5). The definition of these operations in Harp are all partition-based and perform in a collective way with routing optimization. When the operation is performed, each worker invokes the method with a local "table" object, all of which are operated together and considered as one distributed dataset in the collective communication. Only the "push & pull" operation is different. The "push & pull" operation contains two parts, "push" and "pull". Since they are always connected with each other, these two are simply considered as one operation. This operation can simulate Types 3 and 4 simultaneously. Both "push" and "pull" involve two table references: global tables and local tables. The global tables hold a global version of each partition, but the local tables each holds a local version of the global tables. In "push", the partitions are sent to the global table objects without modifying the contents in the local table objects. Similarly, in "pull", the partitions are pulled from the global table objects to the local table objects while the contents in the global

Table 5: Collective Communication Operation Interfaces

Operation Name	Definition
broadcast	The master worker broadcasts all the partitions to the tables on other workers, the remaining workers wait for the partitions to be broadcasted.
reduce	Partitions from all workers are reduced to the master worker (partitions with the same IDs are combined).
allreduce	Partitions from all workers are reduced and received by all workers.
allgather	Partitions from all workers are gathered on all the workers.
regroup	Regroup partitions on all workers based on partition IDs (partitions with the same IDs are combined).
push & pull	Partitions are pushed from local tables to the global table or pulled from the global table to local tables.
rotate	Build a virtual ring topology and shift partitions from a worker to a neighboring worker.

table objects are not modified.

4.4 Mapping Computation Models to Harp Programming Interfaces

Both the "Rotation" and "Allreduce" computation models can be implemented through the Harp programming interfaces with collective communication operations. Model parameters are stored with partitions and tables. To perform the "Rotation" computation model, the "rotate" operation is used. For each round of rotation, users are required to invoke the "rotate" operation a certain number of times equal to the number of workers. Since the model parameters are fully distributed among workers and are shifted among neighboring workers in chunks, there is neither a memory issue of holding a large number of model parameters nor a routing issue of communication.

To perform the "Allreduce" computation model, there are four options of collective communication operations. When the model size is small, each Mapper can have a separate copy of all the parameters. Synchronization can be performed through collective communication operations such as "broadcast-reduce", "allgather", or "allreduce", all of which can efficiently synchronize model copies on all the workers. When the model size increases but can still be held by each machine, synchronization can be performed through "regroup-allgather" to reduce communication overhead. For a large model which cannot be held in the memory of one machine, Harp relies on the "push & pull" operation. Since the "push & pull" operation tries to search for the locality of the model computation dependency and only provides each worker with the model partitions required in lo-

cal computation, this approach can reduce the model parameters required per worker. However, if the local computation requires all the model parameters and they cannot be held by each machine, this solution cannot work. In this case, the parallelization can be solved with two rounds of model rotation. In the first round, each worker accumulates partial computation results based on the model parameters shifted and finally generates local model updates. Then with the second round of rotation, the model updates are collected and applied to the global model. Of course, these "two rounds of rotation" can also work as "allreduce", "regorup-allgather", and "push & pull" operations to synchronize local model copies in the previous cases.

The "Locking" and "Asynchronous" computation models cannot be covered by collective communication operations, but Harp does provide a set of event-driven interfaces, including "getEvent", "waitEvent", and "sendEvent", for designing machine learning algorithms in the "Locking" and "Asynchronous" computation models. However, the "Locking" computation model requires high overhead in distributed locking. From what is observed in the LDA experiments, the "Asynchronous" computation model can be inefficient compared with the "Allreduce" computation model with collective communication operations. Therefore, machine learning algorithm parallelization in Harp focuses on using the "Allreduce" and "Rotation" computation models with collective communication operations.

5 Harp Framework Design and Implementation

In this section, the design and implementation of the Harp framework is described. The ideas of Harp are implemented in an open source library as a Hadoop plug-in. By plugging Harp into Hadoop, the MapCollective programming model with efficient in-memory collective communication for machine learning applications is enabled.

5.1 Layered Architecture

The current Harp plug-in targets Hadoop 2.6.0 (also compatible with 2.7.3). The whole software is written in Java. Through extending the components of the original Hadoop MapReduce framework, MapCollective applications can be run side by side with MapReduce applications (see Figure 13). In Hadoop, a MapCollective application is launched in a way similar to how the MapReduce applications are launched. There is a "Runner" at the client and an "AppMaster" to manage the running states of MapCollective applications. All these components are extended from the original "Runner" and "AppMaster" components in the MapReduce framework. Thus, many functionalities of the original MapReduce framework can be reused in the MapCollective framework, such as input data splitting and Key-Value pairs generating. However, unlike Map tasks in the MapReduce application, all Map tasks have to be kept alive for in-memory communication. As a result, "AppMaster" for a MapCollective application maintains the host addresses of Map tasks. When the Map tasks start running, they use the host address information to establish a communication group. Each task uses a Key-Value reader to read all Key-Value pairs into the memory and an user-implemented "mapCollective" function to process input data in iterations with synchronization through collective communication operations. The whole execution flow is presented in Figure 14.

5.2 Data Interface Methods

Data interfaces have been divided into arrays and objects horizontally and formed into partitions and tables vertically. In Harp, all these are implemented with pool-based memory management. The rationale is that in iterative processing, arrays and objects may be repeatedly used in iterations but filled with different contents. Thus, it is necessary to keep the memory allocation in a cache pool to avoid the Garbage Collection (GC) overhead of repeat memory allocation. Table 6 shows

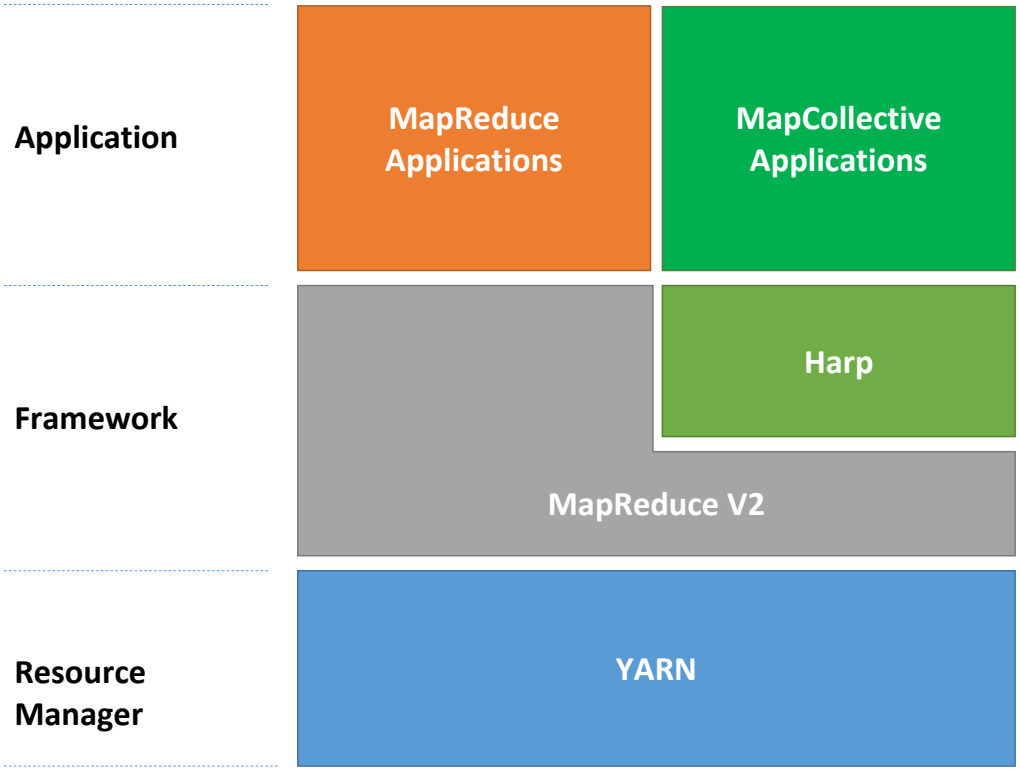


Figure 13: Layered Architecture

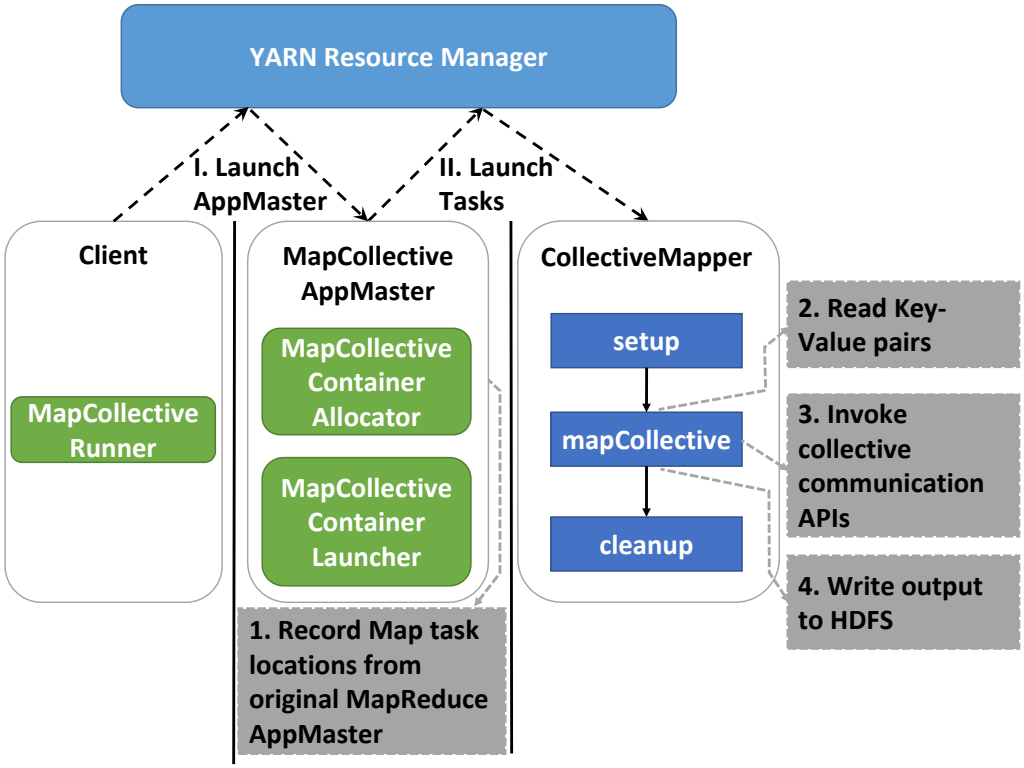


Figure 14: Execution Mechanism of Harp Plug-in

Table 6: Data Interface Methods

Array	Methods
	static create(int len, boolean approximate) - fetch array allocation from the pool. The boolean parameter indicates if the real allocation size can be padded to increase the chance for reuse
	release() - release the array back to the pool
	free() - free the allocation to GC
	get() - get the array body
	start() - get the start index of the array
	size() - get the size of the array
Writable	Methods
	static create(Class<W> clazz) - create an object based on the class
	release() - release the object to the pool
	free() - free the object to GC
	Interfaces
	getNumWriteBytes() - calculate the number of bytes to be serialized
	write(DataOutput out) & **read(DataInput in)** - interfaces for serialization / deserialization
	clear() - clean the fields of the object before releasing to the pool
Partition <P extends Simple>	Constructor & Methods
	Partition(int partitionID, P partition)
	int id() - get the partition ID
	P get() - get the content object of the partition
Table <P extends Simple>	Constructor & Methods
	Table(int tableID, PartitionCombiner<Simple> combiner)
	int getTableID() - user defined table ID
	PartitionCombiner<P> getCombiner() - combiner can combine partitions with the same ID in the table
	PartitionStatus addPartition(Partition<P> partition) - add a partition to the table, return the status to check if the partition is added or combined
	Partition<P> getPartition(int partitionID) - get a partition by ID
	Partition<P> removePartition(int partitionID) - remove a partition from the table

that for both primitive arrays and the "Writable" objects, users can use the "create" method to get a cached data structure from the pool and use the "release" method to return it back to the pool, or use the "free" method to let GC handle the allocation directly. For arrays, a parameter called "approximate" is used in the "create" method. When it is set to true, it allows an array with a larger size to be returned so that the reuse rate of an array allocation is increased. Other operations such as "get", "start", and "size" are provided for accessing the body of the primitive array. To define a class under the "Writable" interface, users need to implement four methods: "getNumWriteBytes", "write", "read" and "clear". "getNumWriteBytes" is used to define the number of bytes an object under this class will use in serialization. The "read" and "write" methods direct object serialization and deserialization. The number of bytes read or written should match the number in "getNumWriteBytes". Finally, the "clear" method directs how the fields in the object is cleaned before returning it to the pool. Based on the arrays and the "Writable" objects, a partition is constructed by a partition ID and a partition body, which can be either an array or a "Writable" object. The "id" method returns the partition ID, and the "get" method returns the partition body. A "Table" is constructed by a partition ID and a combiner. Users can add partitions to the table or get or remove partitions from the table.

5.3 Collective Communication Implementation

For each collective communication operation, a context name and an operation name is defined to identify a collective communication operation. This allows users to group communication operations into different threads and invoke them concurrently. The implementation of each collective communication operation is listed in Table 7.

The "minimum spanning tree" algorithm, "bucket" algorithm, and the "bidirectional exchange" algorithm are all classic MPI collective communication algorithms [36]. The "broadcast" operation is optimized with two algorithms for different sizes of the communication data. If the communication data is small, users are suggested to use the minimum spanning tree algorithm. When the communication data goes large, it is more efficient to use the chain broadcasting algorithm. For the "allreduce" operation, the "bidirectional-exchange" algorithm is designed for the small data. For allreducing large data, the other option is to use "regroup-allgather" operations together or use the "push & pull" operation. Both "regroup" and "rotate" operations are provided with extra options to allow users to decide the mapping between the partitions and the target worker in the data move-

Table 7: Collective Communication Operations

Operation Name	Algorithm	Time Complexity[a]
broadcast	chain	$n\beta$
	minimum spanning tree	$(\log_2 p)n\beta$
reduce	minimum spanning tree	$(\log_2 p)n\beta$
allgather	bucket	$pn\beta$
allreduce	bidirectional exchange	$(\log_2 p)n\beta$
regroup	point-to-point direct sending	$n\beta$
push & pull	point-to-point direct sending plus routing optimization	$n\beta$
rotate	direct sending between neighbors on a ring topology	$n\beta$

[a]Note in "time complexity", p is the number of processes, n is the number of communication data entries per worker, β is the per data entry transmission time, communication startup time is neglected and the time complexity of the "point-to-point direct sending" algorithm is estimated regardless of potential network conflicts.

ment. In the "push & pull" operation, the "push" operation allows the user to decide the mapping between partitions and workers if the related global partition is not found, while the "pull" operation is optimized with broadcasting for the partitions which are required by all the workers.

5.4 Intra-Worker Scheduling and Multi-Threading

The current many-core architecture encourages two-level parallelism. When the number of cores is increasing, it is common to assign one level of parallelism to inter-node processes and another to intra-node threads. In Harp, to embrace two levels of parallelism, each Map task is a process, and they are synchronized through collective communication operations. For multi-threading within the Map processes, users can use schedulers provided by Harp, start Java threads by themselves, and use Java execution services or Java parallel streams.

The two schedulers in Harp are the dynamic scheduler and the static scheduler (see Figure 15). The dynamic scheduler does not distinguish the parallel threads. Each thread keeps fetching inputs from one shared queue and sends the output generated to another shared queue. In this way, the inputs are dynamically scheduled to different threads, and the multi-threaded computation can be balanced. In contrast, for the static scheduler, each thread has its own input and output queues. One thread may submit input objects to other threads' input queue, which enables the message to pass between threads. When retrieving output, users need to specify the queue based on the thread, and

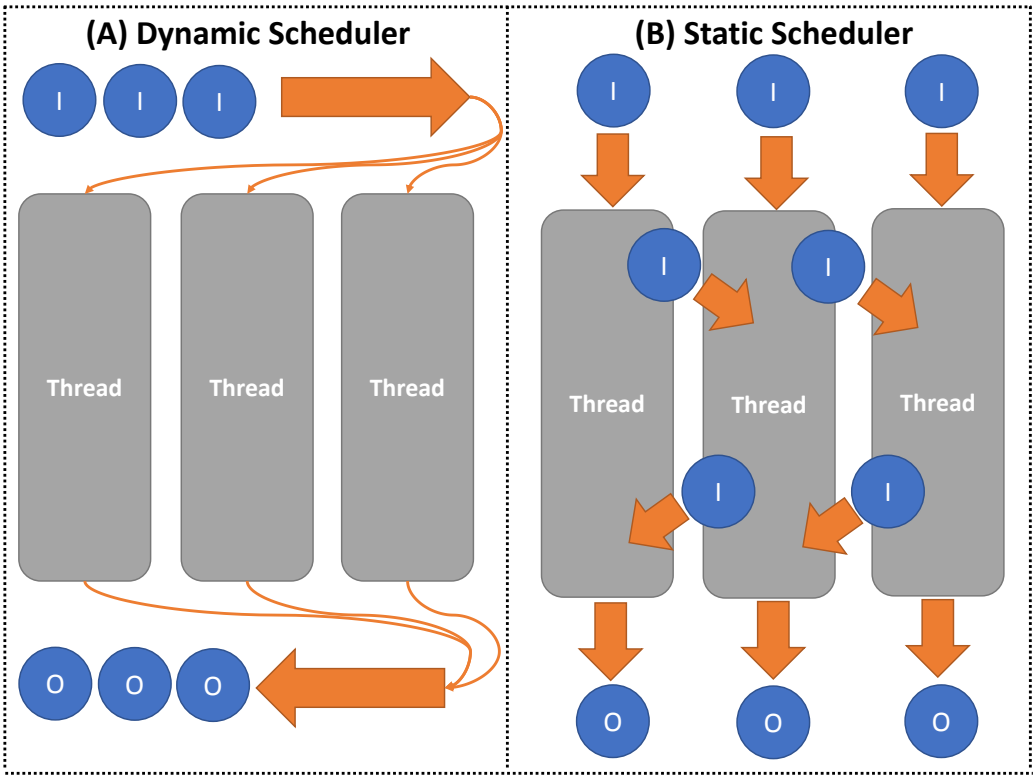


Figure 15: Schedulers for Multi-Threading

thus, the load-balancing of the multi-threaded computation is controlled by the users themselves. Therefore, the philosophy behind these two schedulers are different. The dynamic scheduler simply processes input objects in parallel and seeks load balancing on multi-threading. However, for the static scheduler, it is designed for solving customized execution dependency between threads.

The four computation models can be mapped to the multi-thread level, however, users are required to manage the synchronization between threads. For the "Locking" and "Asynchronous" computation models, users are required to control access to the shared model parameters. For the "Allreduce" computation model, user can spawn multiple threads to perform the computation and then wait for their completion. The "Rotation" computation model is implemented in different ways under the dynamic and static schedulers. When the dynamic scheduler is used, the user has to maintain execution progress and avoid model update conflict. When the static scheduler is used, the user can command each thread to process a set of model parameters and then submit them to another thread. In the new rotation-based solution (see Section 8), the dynamic scheduler is further packaged to be able to perform the model rotation in a simple way, which only requires users to fill the

code of how to update a chunk of model parameters with a chunk of training data entries.

6 Machine Learning Applications on Top of Harp

Many machine learning applications can be built on top of Harp directly. With the computation models and programming interfaces, Harp enables developers to handle complicated model synchronization and program iterative machine learning algorithms productively. Machine learning algorithms that can be implemented on Harp include but are not limited to:

- **Expectation-Maximization Type**
  - K-means Clustering
  - Collapsed Variational Bayesian for topic modeling (e.g. LDA)
- **Gradient Optimization Type**
  - Stochastic Gradient Descent and Cyclic Coordinate Descent for classification (e.g. SVM and Logistic Regression), regression (e.g. LASSO), Collaborative Filtering (e.g. Matrix Factorization)
- **Markov Chain Monte Carlo Type**
  - Collapsed Gibbs Sampling for topic modeling (e.g. LDA)

Before implementing a machine learning application, it is important to know that one machine learning application can be solved by more than one algorithms. For example, K-means Clustering with Lloyd's algorithm [12] is an Expectation-Maximization type of algorithm. However, Mini-Batch K-means [41], though it is still K-means Clustering, is classified as a Gradient Optimization type of algorithm. At the same time, one algorithm can also be used to solve different machine learning applications. For example, Stochastic Gradient Descent can be used in Linear Regression or Matrix Factorization. However, when the application changes, the model contents and the computation dependency of the algorithm are also changed; only the algorithm method itself is kept.

Once an algorithm is selected for implementation, developers need to select the computation model while considering the selection of collective communication operations, the balance of the computation and communication load between processes and threads, and the optimization of per-thread implementation (see Figure 16). The LDA application has shown that model rotation can improve model convergence speed due to the use of the latest model parameters. However, there are also other algorithms which cannot perform direct model update, e.g. Expectation-Maximization type algorithms where model update only happens in the maximization step. All these kinds of

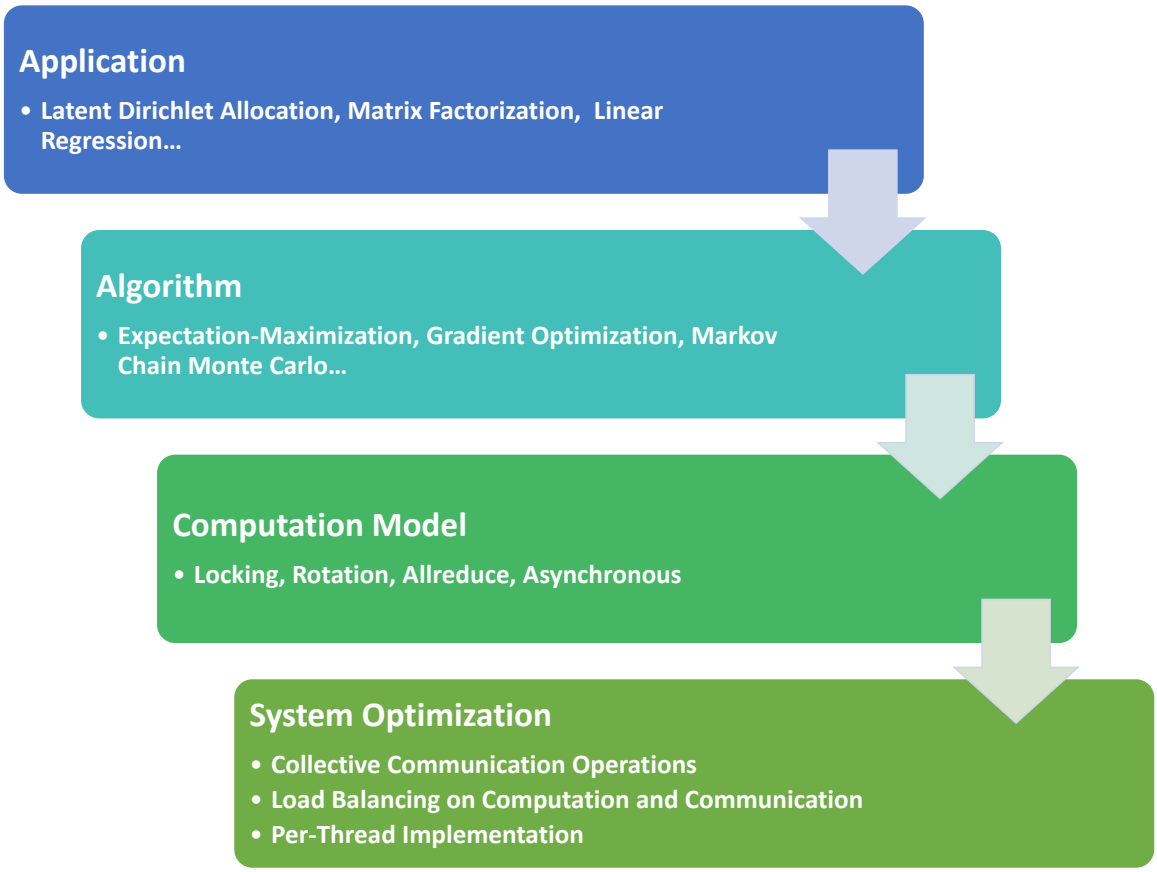


Figure 16: Guidelines of Parallel Machine Learning Application Implementation

Table 8: Machine Learning Application Examples

Application	Model Size	Model Computation Dependency
K-means Clustering	In MBs, but can grow to GBs	All
WDA-SMACOF [13]	In MBs	All
CGS for LDA	From a few GBs to 10s of GBs or more	Partial
SGD for MF		
CCD for MF		

algorithms have to use the "Allreduce" computation model. In this computation model, the parallelization solution needs to select the collective communication operations used for model synchronization. The decision is based on model size and model computation dependency. As what have been discussed in Section 4.4, the "rotate" operation can make the application scalable when the model size increases and when the model computation dependency is dense. However, if the model computation dependency on each worker is sparse, the workers cannot get enough computation by shifting a part of the model parameters. Balancing the load on each worker becomes hard, resulting in huge synchronization overhead. In this case, the "push & pull" operation should be used.

Table 8 shows the characteristics of model computation in some machine learning application examples, including K-means Clustering with Lloyd's algorithm, Scaling by MAjorizing a COmplicated Function with Weighted Deterministic Annealing (WDA-SMACOF), Collapsed Gibbs Sampling for Latent Dirichlet Allocation (CGS for LDA), Stochastic Gradient Descent and Cyclic Coordinate Descent for Matrix Factorization (SGD for MF and CCD for MF). In the following sections, I describe how the first two are solved by the "Allreduce" computation model while the last three are solved by the "Rotation" computation model.

7 The Allreduce-based Solution

I developed two machine learning applications in the "Allreduce" computation model, K-means Clustering and WDA-SMACOF. They are implemented through the "allreduce" and "allgather" operations. By simply keeping the computation load balanced, the efficiency of using MPI-style collective communication operations is directly shown through the speedup.

In all experiments, I tested on Big Red II[13], using the nodes in the "cpu" queue where the maximum number of nodes allowed for job submission is 128. Each node has 32 cores and 64GB memory. The nodes are running in "Cluster Compatibility Mode" and connected with Cray Gemini interconnect. Data Capacitor II (DC2) is for storing the data. File paths on DC2 are grouped into partition files on HDFS to let each Map task read file paths as Key-Value pairs. In all these tests, each node deploys one Map task and utilizes all 32 cores to do multi-threading.

7.1 K-means Clustering

K-means clustering calculates the Euclidean distance between the point vectors (training data) and the centroids (model) in each iteration. At the start of K-means, each worker loads and caches a part of the training points while a single worker needs to prepare initial centroids and use the "broadcast" operation to send data to all other workers. In every iteration, the workers run their own calculations and then use the "regroup-allgather" operations to get the new global centroids.

K-means clustering is tested with two different randomly generated datasets. One is clustering 500 million 3D points into ten thousand clusters, while another is clustering five million 3D points into one million clusters. In the former case, the input data is about 12GB, and the ratio of points to clusters is 50000:1. In the latter case, the input data size is only about 120MB, but the ratio is 5:1. Such a ratio is commonly high in clustering; the low ratio is used in a scenario where the algorithm tries to do fine-grained clustering as classification [42]. The baseline test uses 8 nodes, then scales up to 128 nodes. The execution time is seen in Figure 17(a) and the speedup in Figure 17(b). Since each point is required to calculate the distance from all the cluster centers, the total workload of the two tests is similar. However, due to the cache effect, "five million points and one million centroids" is slower than "500 million points and ten thousand centroids" when the number of nodes is small.

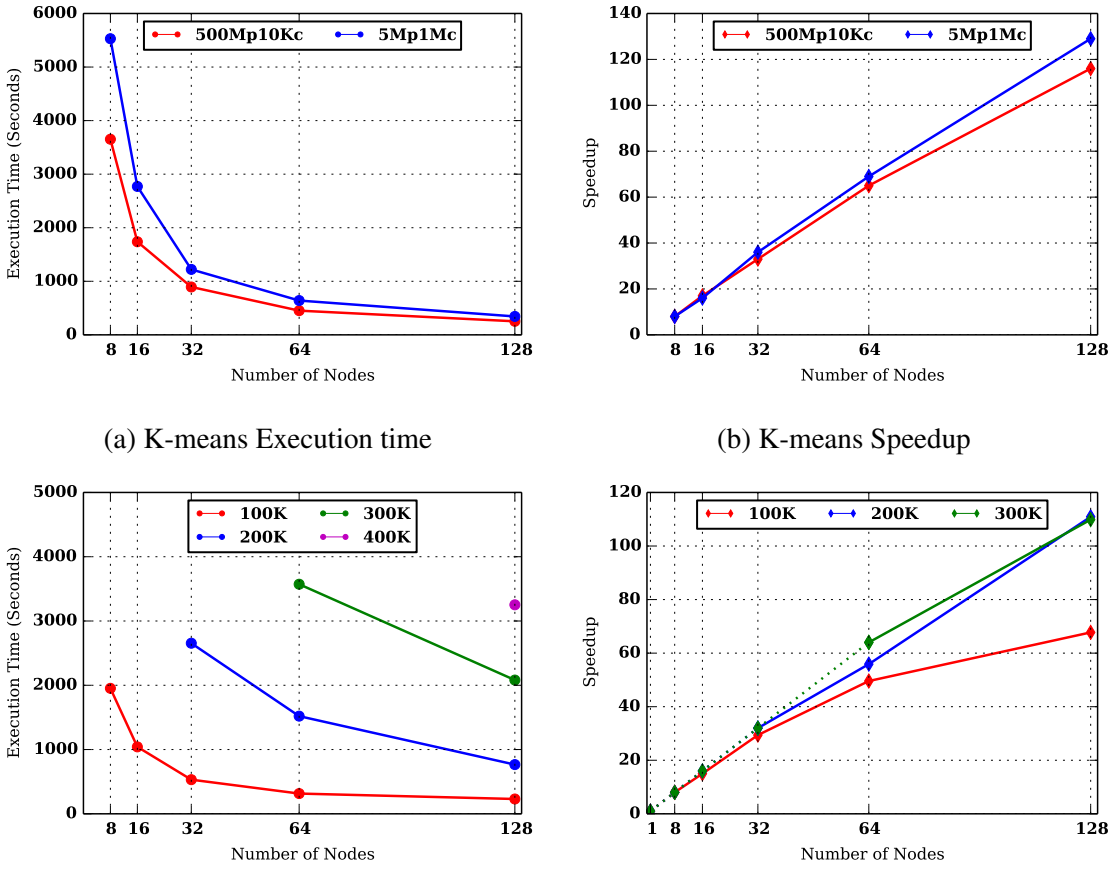
[13] `https://kb.iu.edu/data/bcqt.html`

As the number of nodes increases, however, they draw closer to one another. Assuming there is linear speedup on 8 nodes, the speedup in both test cases is close to linear.

7.2 WDA-SMACOF

SMACOF is a gradient descent type of algorithm used for large-scale multi-dimensional scaling problems. Through iterative stress majorization, the algorithm minimizes the difference between distances from points in the original space and their distances in the new space. WDA-SMACOF improves on the original SMACOF [13]. It uses deterministic annealing techniques to avoid local optima during stress majorization and employs conjugate gradient for the equation solving with a non-trivial matrix to keep the time complexity of the algorithm as $O(N^2)$. WDA-SMACOF has nested iterations. In every outer iteration, the algorithm first does an update on an order N matrix, then performs a matrix multiplication; the coordination values of the points on the target dimension space are calculated through the conjugate gradient process in inner iterations; the stress value of this iteration is determined as the final step. The algorithm is expressed with the "allgather" and "allreduce" operations. In outer iterations, "allreduce" sums the results from the stress value calculation. For inner iterations, the conjugate gradient process uses "allgather" to collect results from matrix multiplication and "allreduce" for those from inner product calculations.

The WDA-SMACOF algorithm runs with different problem sizes including 100K, 200K, 300K and 400K points. Each point represents a gene sequence in a dataset of 454 representative pyrosequences from spores of known AM fungal species [43]. Since the training data is the distance matrix of points and related weight matrices, the total size of the input data is in quadratic growth. It is about 140GB for the 100K problem, 560GB for 200K, 1.3TB for 300K, and 2.2TB for 400K. Due to memory limitations, the minimum number of nodes required to run the application is 8 for the 100K problem, 32 for the 200K, 64 for 300K, and 128 for 400K. The execution time and speedup are seen in Figures 17(c) and 17(d). Since running each input on a single machine is impossible, the minimum number of nodes required to run the job is selected as the base to calculate parallel efficiency and speedup. In most cases, the efficiency values are very high. The only point that has low efficiency is the 100K problem on 128 nodes. This is a standard effect in parallel computing where the small problem size reduces computing time compared to communication, which in this case has an overhead of about 40% of the total execution time.



(a) K-means Execution time

(b) K-means Speedup

(c) WDA-SMACOF Execution time

(d) WDA-SMACOF Speedup

Figure 17: The Performance of K-means and WDA-SMACOF

8 The Rotation-based Solution

This section is based on a published paper [21]. LDA and MF have been successfully applied to big data within various domains. For example, Tencent uses LDA for search engines and online advertising [34] while Facebook[14] uses MF to recommend items to more than one billion people. With tens of billions of training data entries and billions of model parameters, these applications can help data scientists gain a better understanding of big data. However, the growth of data size and model size makes it hard to deploy these applications in a way that scales to peoples' needs. Previous analysis on computation models has shown that the advantage of model rotation comes from maximizing the effect of parallel model updates for algorithm convergence while minimizing the overhead of communication for scaling. In this section, a new model rotation-based solution is designed to parallelize three algorithms: CGS for LDA, SGD and CCD for MF.

Model rotation has been applied in the implementations of these applications before. In LDA, F. Yan et al. implement CGS on a GPU [44]. In MF, DSGD++ [45] and NOMAD[15] [46] use model rotation in SGD for MF in a distributed environment while LIBMF [47] applies it to SGD on a single node through dynamic scheduling. Another work, Petuum Strads [28, 29, 30], supplies a general parallelism solution called "model parallelism" through "schedule-update-aggregate" interfaces. This framework implements CGS for LDA using model rotation but does not use it in CCD for MF[16]. Instead it uses the "allgather" operation to collect model parameters without using model rotation. Thus Petuum CCD cannot be applied to big model applications due to the memory constraint. The new model rotation-based solution tries to solve the big model problem in the LDA and MF applications. Though CGS for LDA has been discussed in Section 2.3, the proposed parallelization solution does not focus only on communication optimization. In this work, high level programming interfaces for model rotation are provided. Further optimizations include pipelining to reduce communication overhead and dynamically controlling the time point of the model rotation for load balancing. Through comparing with state-of-the-art implementations running side-by-side on the same cluster, performance results of the Harp CGS, SGD and CCD implementations show that they achieve fast model convergence speed and high scalability.

[14] https://code.facebook.com/posts/861999383875667/recommending-items-to-more-than-a-billion-people

[15] http://bikestra.github.io/

[16] https://github.com/petuum/strads/tree/master/apps/

Table 9: Sequential Algorithm Pseudo Code of SGD and CCD for MF

SGD Algorithm for MF

Input: training matrix V, the number of features K, regularization parameter λ, learning rate ϵ

Output: row related model matrix W and column related model matrix H

Initialize W, H to *UniformReal*$(0, \frac{1}{\sqrt{K}})$
repeat
  for random $V_{ij} \in V$ **do**
    $error = W_{i*}H_{*j} - V_{ij}$
    $W_{i*} = W_{i*} - \epsilon(error \cdot H_{*j}^{\intercal} + \lambda W_{i*})$
    $H_{*j} = H_{*j} - \epsilon(error \cdot W_{i*}^{\intercal} + \lambda H_{*j})$
  end for
until convergence

CCD Algorithm for MF

Input: training matrix V, the number of features K, regularization parameter λ

Output: row related model matrix W and column related model matrix H

Initialize W, H to *UniformReal*$(0, \frac{1}{\sqrt{K}})$
Initialize residual matrix R to $V - WH$
repeat
  for $V_{*j} \in V$ **do**
    for $k = 1$ **to** K **do**
      $s^* = \frac{\sum_{i \in V_{*j}} (R_{ij} + H_{kj}W_{ik})W_{ik}}{\sum_{i \in V_{*j}} (\lambda + W_{ik}^2)}$
      $R_{ij} = R_{ij} - (s^* - H_{kj})W_{ik}$
      $H_{kj} = s^*$
    end for
  end for
  for $V_{i*} \in V$ **do**
    for $k = 1$ **to** K **do**
      $z^* = \frac{\sum_{j \in V_{i*}} (R_{ij} + W_{ik}H_{kj})H_{kj}}{\sum_{j \in V_{i*}} (\lambda + H_{kj}^2)}$
      $R_{ij} = R_{ij} - (z^* - W_{ik})H_{kj}$
      $W_{ik} = z^*$
    end for
  end for
until convergence

8.1 Algorithms

The sequential algorithm of CGS for LDA has been listed in Table 2. CGS learns the model parameters by going through the tokens in a collection of documents D and computing the topic assignment Z_{ij} on each token $X_{ij} = w$ by sampling from a multinomial distribution of a conditional probability of Z_{ij}: $p\left(Z_{ij} = k \mid Z^{\neg ij}, X_{ij}, \alpha, \beta\right) \propto \frac{N_{wk}^{\neg ij} + \beta}{\sum_w N_{wk}^{\neg ij} + V\beta} \left(M_{kj}^{\neg ij} + \alpha\right)$. Here superscript $\neg ij$ means that the corresponding token is excluded. V is the vocabulary size. N_{wk} is the current token count of the word w assigned to topic k in K topics, and M_{kj} is the current token count of the topic k assigned in the document j. α and β are hyperparameters. The model includes Z, N, M and $\sum_w N_{wk}$. When $X_{ij} = w$ is computed, some elements in the related row N_{w*} and column M_{*j} are updated. Therefore dependencies exist among different tokens when accessing or updating N and M model matrices.

The sequential algorithms of SGD and CCD for MF are listed in Table 9. MF decomposes a

$m \times n$ matrix V (training dataset) to a $m \times K$ matrix W (model) and a $K \times n$ matrix H (model). The SGD algorithm learns the model parameters by optimizing the object loss function composed by a squared error and a regularizer (using L_2 regularization). When an element V_{ij} is computed, the related row vector W_{i*} and column vector H_{*j} are updated. The gradient calculation of the next random element $V_{i'j'}$ depends on the previous updates on $W_{i'*}$ and $H_{*j'}$. CCD also solves the MF application, but unlike SGD, the model update order first goes through all the rows in W and then the columns in H, or all the columns in H first and then rows in W. The model update inside each row of W or column of H goes through feature by feature.

CGS, SGD and CCD can all be implemented by the "Allreduce/Asynchronous" computation models [29, 35, 45, 48, 49]. Due to the fact that each model update is only related to a part of the model parameters, the parallelization of these three algorithms can be performed through the "Rotation" computation model, which has been proven to perform better [29, 30, 45]. However, model rotation may result in high synchronization overhead in these algorithms due to the dataset being skewed, generating unbalanced workload on each worker [19, 47]. Therefore, the completion of an iteration has to wait for the slowest worker. If the straggler acts up, the cost of synchronization becomes even higher. In the rotation-based solution, this problem is taken into consideration to minimize the synchronization overhead. It has been noted that in CGS and SGD, the model parameters for update can be randomly selected: CGS by its nature supports random scanning on model parameters [50] while SGD allows random selection on model parameters for updating through randomly selecting entries from the training dataset. Due to this algorithm feature, it is possible to dynamically control the time point of model synchronization for load-balancing.

8.2 Programming Interface and Implementation

This subsection describes the model rotation solution based on the Harp MapCollective framework and demonstrates how model rotation is applied to three algorithms.

8.2.1 Data Abstraction and Execution Flow

The structure of the training data can be generalized as a tensor. For example, the dataset in CGS is a document-word matrix. In SGD, the dataset is explicitly expressed as a matrix. When it is applied to recommendation systems, each row of the matrix represents a user and each column an item;

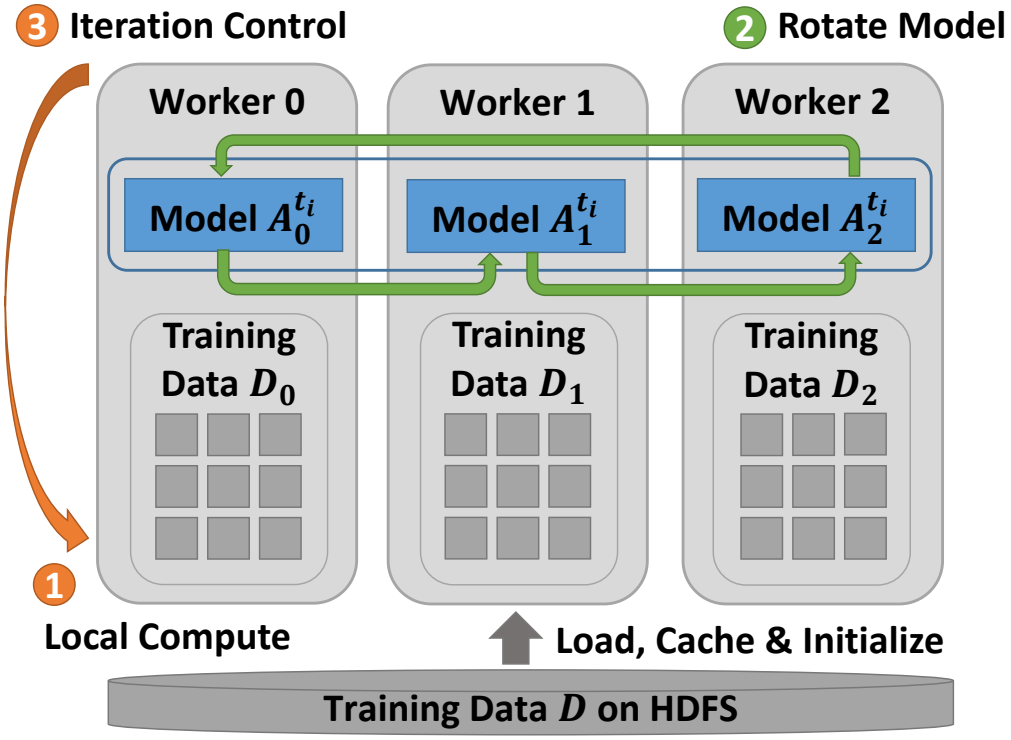


Figure 18: Execution Flow of Model Rotation

thus every element represents the rating of a user to an item. In these matrix structured training data, each row has a row-related model parameter vector as does each column. For quickly visiting each entry and related model parameters, indices are built on the training matrix's row IDs or the column IDs. Based on the model parameter settings, the number of elements per vector can be very large. As a result, both row-related and column-related model structures might be large matrices. In CGS and SGD, the model update function allows the training data to be split by rows or columns so that one model matrix (with regards to the matching row or column of training data) is cached with the data, leaving the other to be rotated. For CCD, the model update function requires both the row-related model matrix W and the column-related model matrix H to be rotated. The model for rotation is abstracted through the partitions and tables in Harp. In CGS, each partition holds a list of a word's topic counts. In SGD, each partition holds a column's related model parameter vector. In CCD, each partition holds a feature dimension's parameters of all the rows or columns.

Model rotation is expressed with the "rotate" operation in Harp (see Figure 18). By default, the operation sends the model partitions to the next neighbor and receives the model partitions

Table 10: Pseudo Code of Model Rotation

Input: P workers, data D, model A^0, the number of iterations T
Output: A^t

```
1: parallel for worker p ∈ [1, P] do
2:   for t = 1 to T do
3:     for i = 1 to P do
4:       A_{p'}^{t_i} = F(D_p, A_{p'}^{t_{i-1}})
5:       rotate A_{p'}^{t_i}
6:     end for
7:   end for
8: end parallel for
```

from the last neighbor in a predefined ring topology of workers. An advanced option is that the ring topology can be dynamically defined before performing model rotation. For local computation inside each worker, they are simply programmed through an interface of "schedule-update". A scheduler employs a user-defined function to maintain a dynamic order of model parameter updates and avoid the update conflict. Since the local computation only needs to process the model obtained during the rotation without considering the parallel model updates from other workers, the code of a parallel machine learning algorithm can be modularized as a series of actions of performing computation and rotating model partitions (see Table 10).

8.2.2 Pipelining and Dynamic Rotation Control

The model rotation operation is wrapped as a non-blocking call so that the efficiency of model rotation can be optimized through pipelining. The distributed model parameters are divided on all the workers into two sets A_{*a} and A_{*b} (see Figure 19). The pipelined model rotation is conducted in the following way: all the workers first compute Model A_{*a} with related local training data. Then they start to shift A_{*a}, and at the same time they compute Model A_{*b}. When the computation on Model A_{*b} is completed, it starts to shift. All workers wait for the completion of corresponding model rotations and then begin computing model updates again. Therefore the communication is overlapped with the computation.

Since the model parameters for update can be randomly selected, dynamic control on the invocation of model rotation is allowed based on the time spent and the number of data entries trained during the time period. (see Figure 20). In CGS for LDA and SGD for MF, assuming each worker caches rows of data and row-related model parameters and obtains column-related model param-

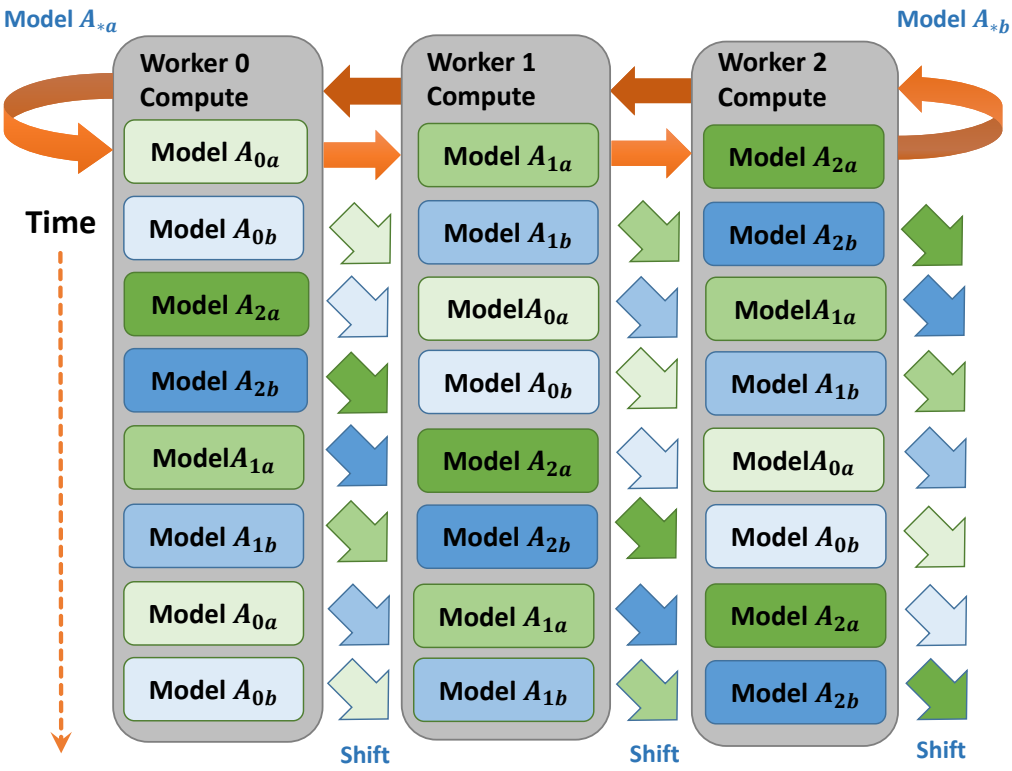


Figure 19: Pipelining

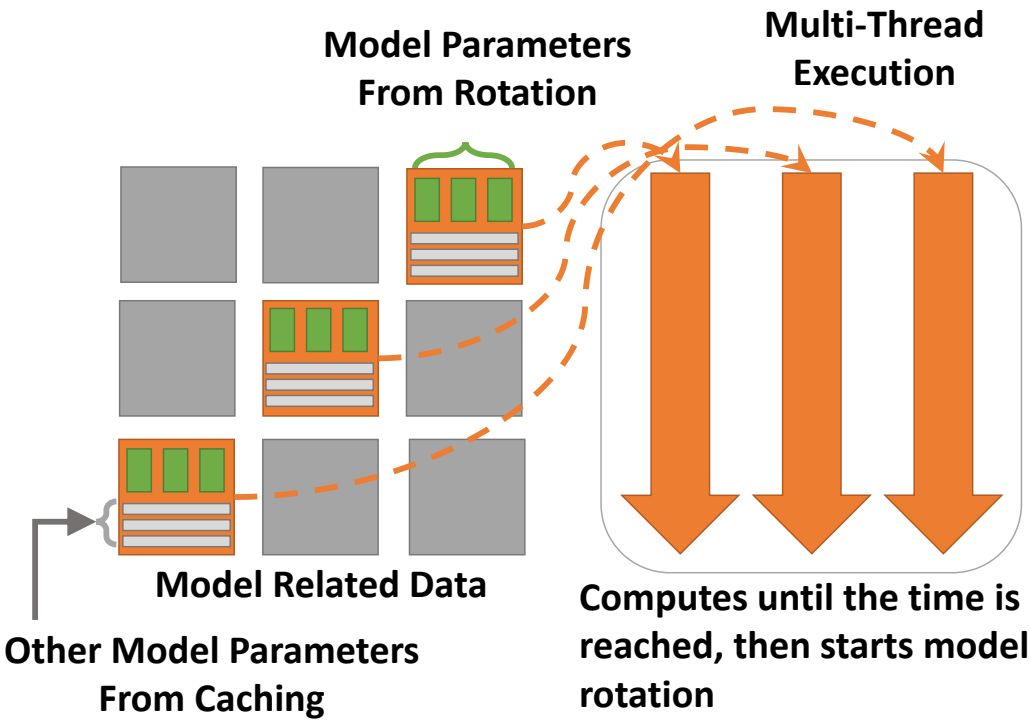


Figure 20: Dynamic Rotation Control

eters through rotation, it then selects related training data to perform local computation. The data and model are split into small blocks which the scheduler randomly selects for model update while avoiding the model update conflicts on the same row or column. Once a block is processed by a thread, it reports the status back to the scheduler. Then the scheduler searches another free block and dispatches to an idle thread. A timer is set to oversee the training progress. When the designated time arrives, the scheduler stops dispatching new blocks, and the execution ends. With the same length of execution time per worker, the computation is load-balanced, and the synchronization overhead is minimized. In the iterations of model rotations, the time length is further adjusted based on the amount of data entries processed. When the multi-threaded local computation progresses, the blocks on some rows or columns may have all been processed, and then only a few blocks from other rows are still left, making the CPU utilization be low. To keep the load-balancing between threads, the execution time length is adjusted to a number that is able to keep most threads running and the communication time overlapped. Usually, the adjusted time length allows each worker to process about half of the local training data entries. Since the algorithms allow data entries to be randomly selected, eventually all the data entries can be processed so that algorithm correctness is guaranteed. When the load-balancing is kept, the throughput in the parallelization is significantly improved which results in fast model convergence speed.

However, in CCD, the algorithm dependency does not allow us to dynamically control model rotation. The reason is that in each iteration of CCD, each model parameter is only updated once by the local training data, and the communication overhead of rotating W and H is high. Using dynamic control cannot save much of the execution time per iteration and results in incomplete model updates, which reduces the model convergence speed.

8.2.3 Algorithm Parallelization

The details of each algorithm's parallelization solution are described as below:

CGS When parallelizing CGS with model rotation, the training data is split by documents. As a result, the document-topic model matrix is partitioned by documents while the word-topic model matrix is rotated among workers. On each worker, documents are partitioned and an inverted index is used to group each partitions' tokens by word. The word-topic matrix owned by the worker is also split into partitions. Thus the training tokens can be selected in blocks, trained and updated

with the related model parameters. Because the computation time per token changes as the model converges [19], the amount of tokens which can be trained during a time period grows larger. As a result, an upper bound and a lower bound are kept for the amount of tokens trained between two invocations of the "rotate" operation.

SGD Both W and H are model matrices. Assuming $n < m$, then V is regrouped by rows, W is partitioned with V, and H is the model for rotation. The ring topology for rotation is randomized per iteration for accelerating model convergence. For pipelining and load-balancing, the execution time length is adjusted to a number to be able to process about 60% of training elements per iteration.

CCD Both W and H are model matrices. Due to the fact that model update on a row of W needs all the related elements in the same row and model update on a column of H needs all the related data entries in the column, the training data is duplicated so that one is regrouped by rows and another is regrouped by columns among workers. However, the model update on each feature dimension is still independent. Then W and H are split by features and distributed across workers. Both W and H are rotated for parallel updating of each feature vector in W and H.

8.3 Experiments

In this subsection, the Harp rotation-based solutions are compared with other state-of-the-art implementations. Experiment settings are described first, then the LDA and MF experiment results are used to show that the Harp implementations have high performance and scalability.

8.3.1 Experiment Settings

For LDA and MF applications, each has one small dataset and one big dataset. The datasets and related training parameters are presented in Table 11. Two big datasets are generated from the same source "ClueWeb09", while two small datasets are generated from Wikipedia. With different types of datasets, the effectiveness of the model rotation solution is shown by the model convergence speed. In LDA, the model convergence speed is evaluated with respect to model likelihood, which is a value calculated from the word-topic model matrix. In MF, model convergence speed is evaluated by the value of the Root Mean Square Error (RMSE) calculated on the test dataset, which is from the original matrix V but separate from the training dataset. The scalability of implementations is examined based on the processing throughput of the training data entries.

Table 11: Training Datasets

LDA Dataset	**Documents**	**Words**	**Tokens**	**CGS Training Parameters**	
clueweb1	76163963	999933	29911407874	$K = 10000$, $\alpha = 0.01$, $\beta = 0.01$	
enwiki	3775554	1000000	1107903672		
MF Dataset	**Rows**	**Columns**	**Elements**	**SGD Training Parameters**	**CCD Training Parameters**
clueweb2	76163963	999933	15997649665	$K = 2000$, $\lambda = 0.01$, $\epsilon = 0.001$	$K = 120$, $\lambda = 0.1$
hugewiki	50082603	39780	3101135701	$K = 1000$, $\lambda = 0.01$, $\epsilon = 0.004$	

Experiments are conducted on a 128-node Intel Haswell cluster and a 64-node Knights Landing cluster at Indiana University. In the Intel Haswell cluster, 32 nodes each has two 18-core Xeon E5-2699 v3 processors (36 cores in total), and 96 nodes each has two 12-core Xeon E5-2670 v3 processors (24 cores in total). All the nodes have 128 GB memory and are connected by QDR InfiniBand. In the tests, JVM memory is set to "-Xmx120000m -Xms120000m", and IPoIB is used for communication. In the Intel Knights Landing cluster, 48 nodes each have one 68-core Xeon Phi 7250F, and 16 nodes each have one 72-core Xeon Phi 7290F. All the nodes have 192 GB memory and are connected by Intel Omni-Path Fabric. In the tests, JVM memory is set to "-Xmx180000m -Xms180000m", and IP on Omni-Path Fabric is used for communication.

8.3.2 LDA Performance Results

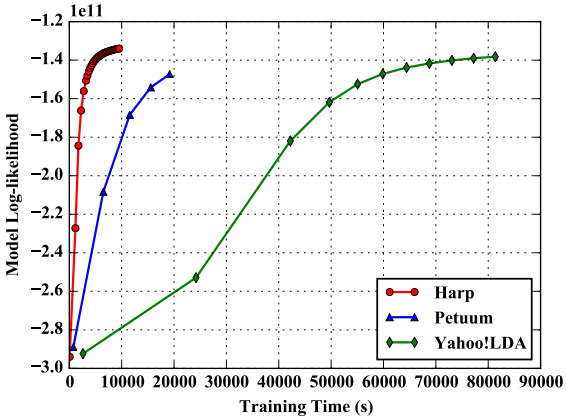
Harp CGS implementation is compared with Petuum LDA and Yahoo! LDA. The implementations are tested on Haswell machines (with two 18-core Xeon E5-2699 v3) and Knights Landing machines (with one 68-core Xeon Phi 7250F) separately. The test plan is shown in Table 12.

In Figure 21, the results on two different datasets, machines and scales all show that Harp consistently outperforms Petuum LDA (by about two to four times). Yahoo! LDA is not tested on "clueweb1" with Knights Landing 12×60 due to the extreme slow performance (see Figure 21(b)). The remaining three settings are more than ten times slower than Harp LDA (see Figure 21(a)(c)(d)). The total number of threads used on the Haswell machines is equal to the number of threads on the Knights Landing machines. However, because each core of Knights Landing is much weaker than

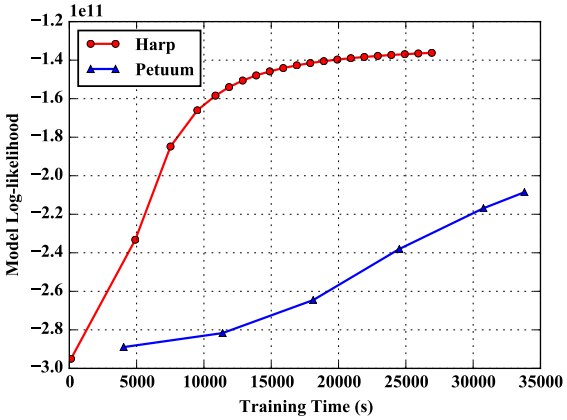
Table 12: Test Plan

Dataset	Node Type	
	two Xeon E5 2699 v3 Haswell (each node uses 30 Threads)	**one Xeon Phi 7250F Knights Landing (each node uses 60 Threads)**
clueweb1	24[a]	12
clueweb1	18, 24, 30	12, 24, 36
enwiki	10	5
enwiki	2, 4, 8, 16	2, 4, 8, 16

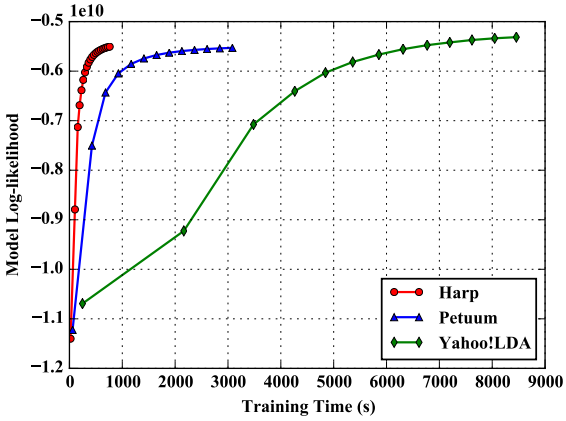
[a]The number of nodes is used.



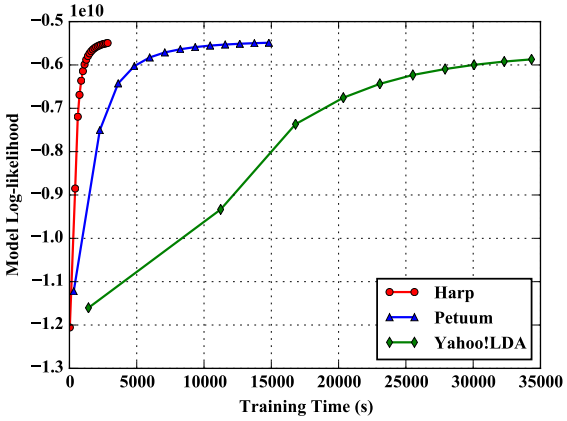
(a) "clueweb1", Haswell, 24 × 30



(b) "clueweb1", Knights Landing, 12 × 60



(c) "enwiki", Haswell, 10 × 30



(d) "enwiki", Knights Landing, 5 × 60

Figure 21: CGS Model Convergence Speed

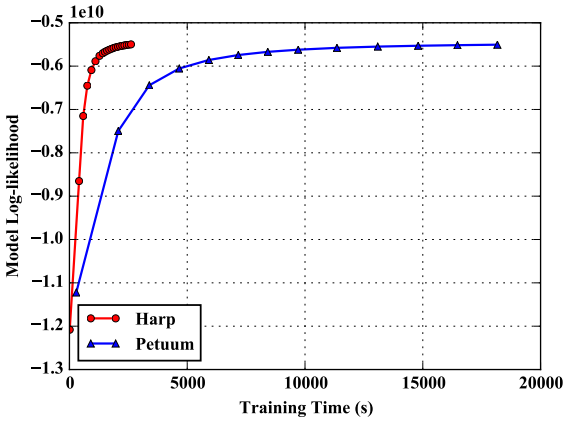
a core of Haswell, the results on Knights Landing are about four times slower than the results on Haswell.

The scaling tests is performed with the "enwiki" dataset. Usually the scalability is evaluated based on the time spent per iteration. However, because the model likelihood values achieved by the same amount of model updates are all different in these implementations, the scalability is examined by the convergence speed and the throughput together. In Figure 22, the results show that on different scales, Harp LDA runs faster than Petuum LDA and Yahoo! LDA on Haswell machines (due to the extreme slowness, results of Yahoo! LDA on two nodes is not available). Figure 22(e) shows that Harp LDA has much higher throughput of token sampling than Petuum LDA for the first 200 seconds on all the scales. The throughput difference between Harp LDA and Petuum LDA reduces a little bit when scaling to 16 nodes. This is because in this setting each iteration of Harp LDA only takes a few seconds with little computation. Similar results are shown on the Knights Landing machines (see Figure 23).

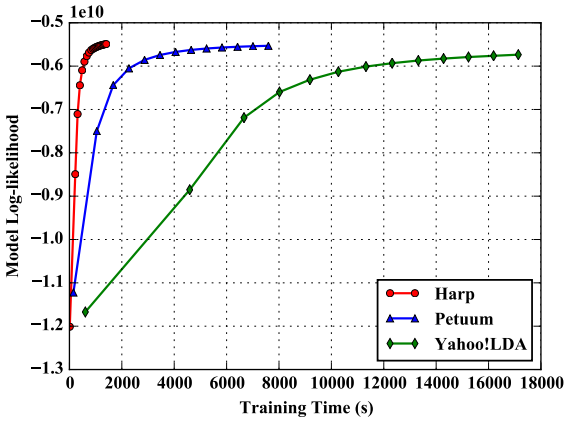
Java is commonly considered to be slower than C++, but Harp LDA using Java 8 shows higher performance than C++ implementations. After detailed comparisons between code implementations, the optimizations of the Harp LDA can be summarized as below:

- Apply high-performance parallel computation model
- Dynamically control model synchronization for load-balancing
- Optimize loops for token sampling
- Utilize efficient data structures
- Cache intermediate results in token sampling

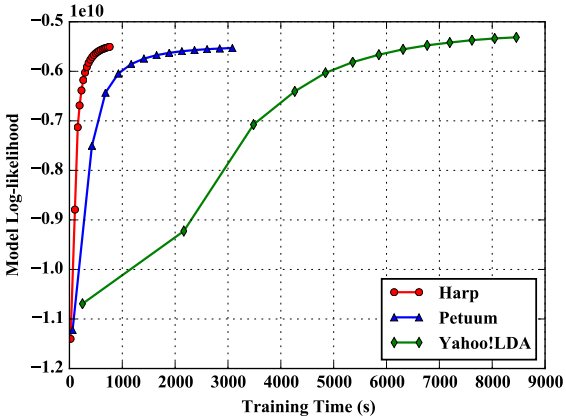
The first two techniques help the Harp LDA achieve excellent parallel performance. The remaining three improve the performance of the sampling procedure in each thread. The time complexity of sampling each token is $\mathcal{O}(\sum_k \mathbb{1}(N_{wk} \neq 0) + \sum_k \mathbb{1}(N_{kj} \neq 0))$. Since the topics of the tokens are randomly initialized and K is 10000, the initial $\sum_k \mathbb{1}(N_{wk} \neq 0)$ on each word is a very large number close to K while $\sum_k \mathbb{1}(N_{kj} \neq 0)$ on each document is a relative small number because each document only contains no more than 1000 tokens. Since each word has more topics than each document, the loop in the sampling procedure goes through each word and then through each document. For the same reason, Harp LDA indexes the words' topics for fast topic searching but uses linear search for documents' topic searching. Topic counts are stored in primitive arrays and



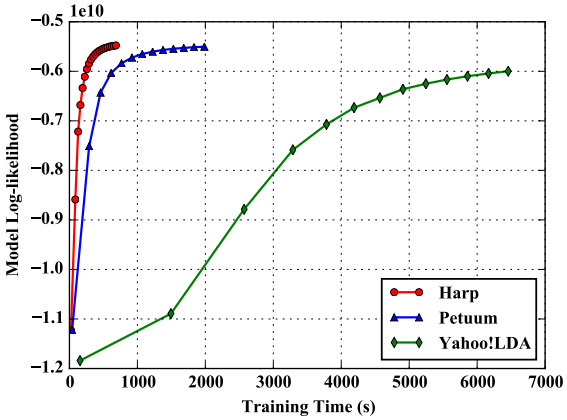
(a) 2 × 30



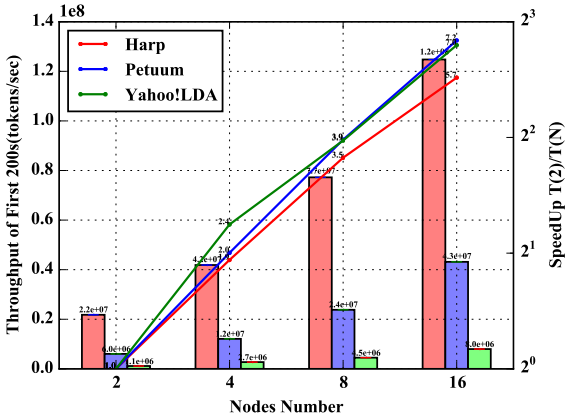
(b) 4 × 30



(c) 8 × 30

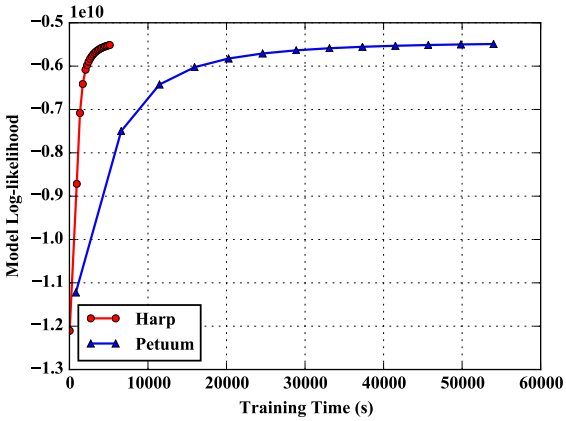


(d) 16 × 30

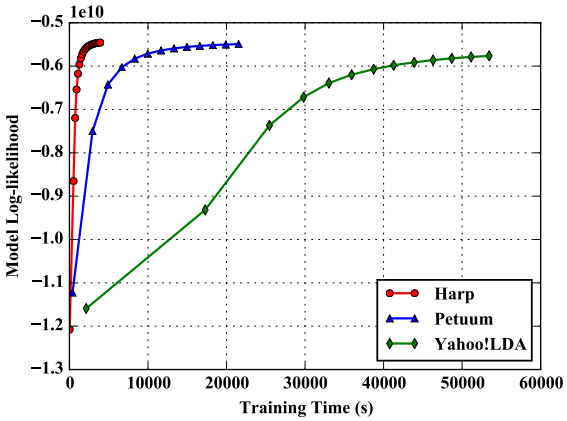


(e) Throughput Speedup

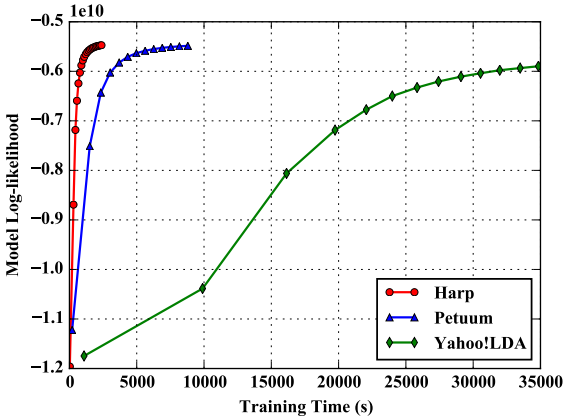
Figure 22: Speedup on “enwiki”, Haswell



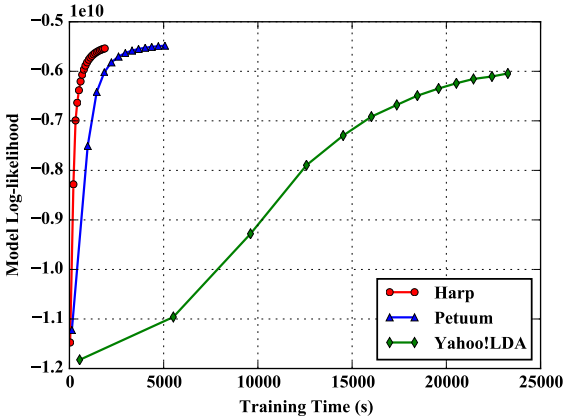
(a) 2 × 60



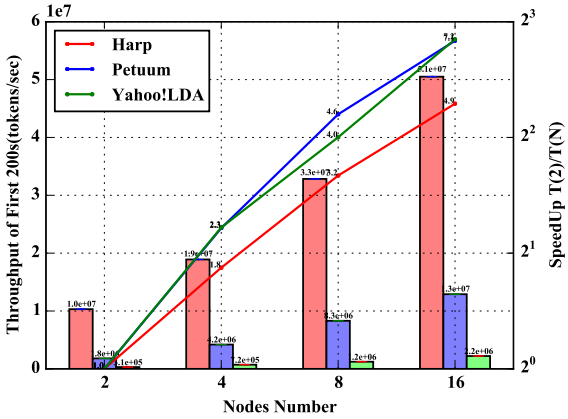
(b) 4 × 60



(a) 8 × 60



(b) 16 × 60



(e) Throughput Speedup

Figure 23: Speedup on “enwiki”, Knights Landing

Table 13: Test Plan

Dataset	Node Type	
	Xeon E5 2699 v3 (each uses 30 Threads)	**Xeon E5 2670 v3 (each uses 20 Threads)**
clueweb2	Harp SGD vs. NOMAD (30) Harp CCD vs. CCD++ (30)	Harp SGD vs. NOMAD (30, 45, 60) Harp CCD vs. CCD++ (60)
hugewiki	Harp SGD vs. NOMAD (10) Harp CCD vs. CCD++ (10)	

sorted from high to low to reduce the time complexity of sampling on a probability distribution. In addition to these optimizations, caching is also used to avoid repeat calculations. When sampling multiple tokens with the same word and document, the topic probabilities calculated for the first token are reused for the following tokens.

8.3.3 MF Performance Results

In MF, Harp SGD is compared with NOMAD. Later Harp CCD is compared to CCD++[17]. Note that both NOMAD and CCD++ are implemented in C++ while Harp SGD and CCD are implemented in Java 8, so it is quite a challenge to exceed them in performance. NOMAD uses MPICH2[18] for inter-node processes and Intel Thread Building Blocks[19] for multi-threading. In NOMAD, MPI "send" and "receive" operations are used for communication, but the destination of model shifting is randomly selected without following a ring topology. CCD++ also uses MPICH2 for inter-node collective communication operations but OpenMP[20] for multi-threading. CCD++ uses a parallelization method different from either Petuum CCD's "allgather" implementation or Harp's model rotation implementation. It allows parallel update on different parameter elements in a single feature vector of W and H. In such a way, only one feature vector in W and one feature vector in H are "allgathered". Thus there is no memory constraint in CCD++ compared with Petuum CCD.

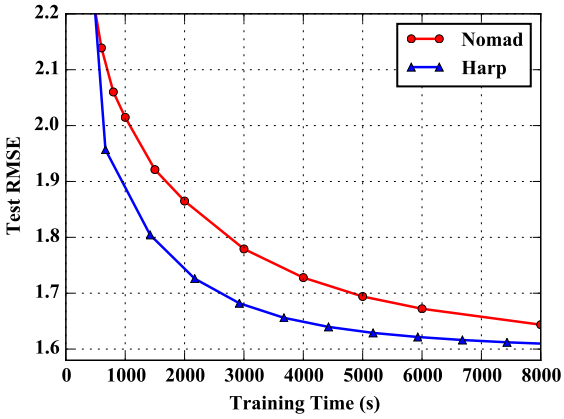
The implementations are tested on two types of machines separately (see Table 13). For the "hugewiki" dataset, 10 Xeon E5-2699 v3 nodes each with 30 threads are used, while the "clueweb2" dataset are undertaken by 30 Xeon E5-2699 v3 nodes and 60 Xeon E5-2670 nodes to compare the model convergence speed among different implementations. The scalability with 30, 45, and 60

[17] `http://www.cs.utexas.edu/~rofuyu/libpmf/`
[18] `http://www.mpich.org`
[19] `https://www.threadingbuildingblocks.org`
[20] `http://openmp.org/wp`

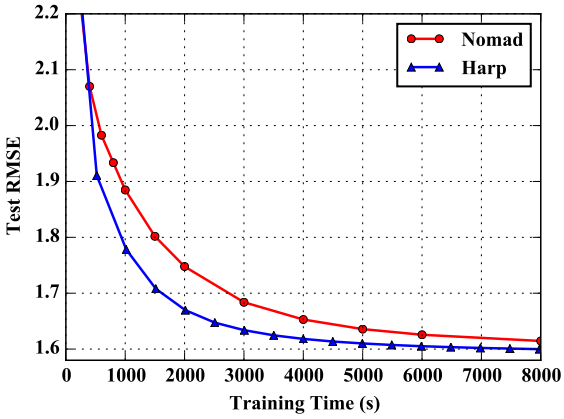
Xeon E5-2670 v3 nodes is further examined each with 20 threads.

In SGD, Harp SGD also converges faster than NOMAD. On "clueweb2", with 30×30 Xeon E5-2699 v3 nodes, Harp is 58% faster, and with 60×20 Xeon E5-2670 v3 nodes, Harp is 93% faster when the test RMSE value converges to 1.61 (see Figure 24(a)(b)(d)). The difference in the convergence speed increases because the random shifting mechanism in NOMAD becomes unstable when the scale goes up. The "hugewiki" dataset is tested on 10×30 Xeon E5-2699 v3 nodes, and the result remains that Harp SGD is faster than NOMAD (see Figure 24(c)). The scalability of SGD implementations is further evaluated with the throughput on the number of training matrix elements processed (see Figure 24(e)). In SGD, the time complexity of processing a training element is $\mathcal{O}(K)$ for all the elements in V; as such this metric is suitable for evaluating the performance of SGD implementations on different scales. Figure 24(e) shows that the throughput of Harp at 1000s achieves $1.7\times$ speedup when scaling from 30 to 60 nodes. Meanwhile, NOMAD only achieves $1.5\times$ speedup. The throughput of NOMAD on three scales are all lower than Harp.

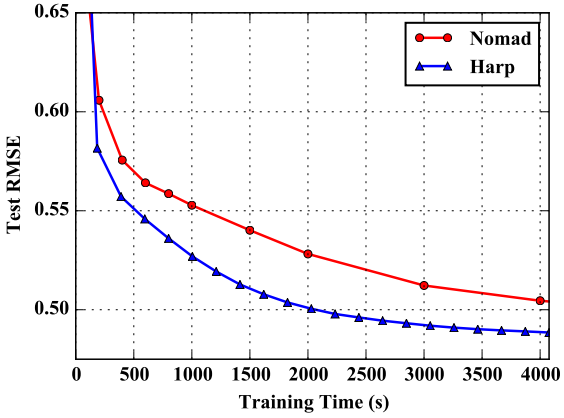
In CCD, the model convergence speed is again tested on the "clueweb2" and "hugewiki" datasets (see Figure 25(a)(b)(c)). The results show that Harp CCD also has comparable performance with CCD++. Note that CCD++ uses a different model update order, so the convergence rate based on the same number of model update counts is different from Harp CCD. However, the tests on "clueweb2" reveal that with 30×30 Xeon E5-2670 v3 nodes, Harp CCD is 53% faster than CCD++, and with 60×20 Xeon E5-2699 v3 nodes, Harp CCD is 101% faster than CCD++ when the test RMSE converges to 1.68 (see Figure 25(d)).



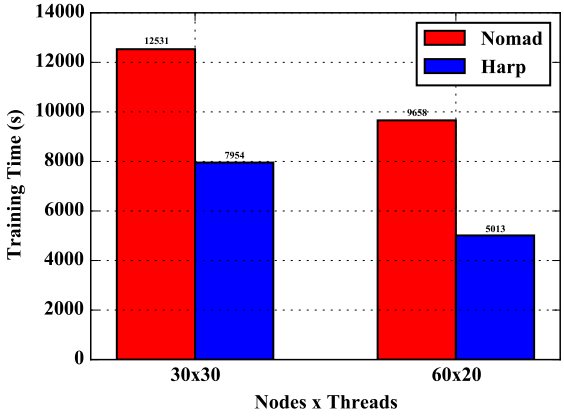
(a) “clueweb2”, 30x30



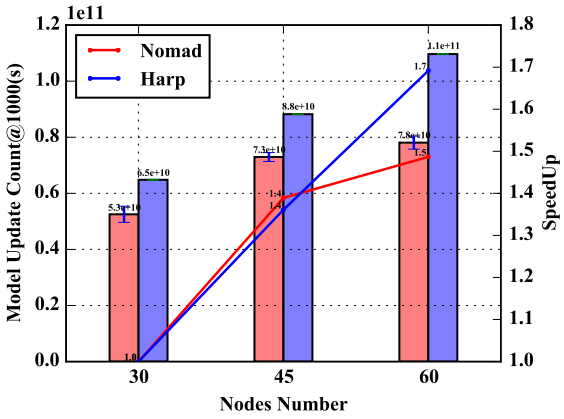
(b) “clueweb2”, 60x20



(c) “hugewiki”, 10x30

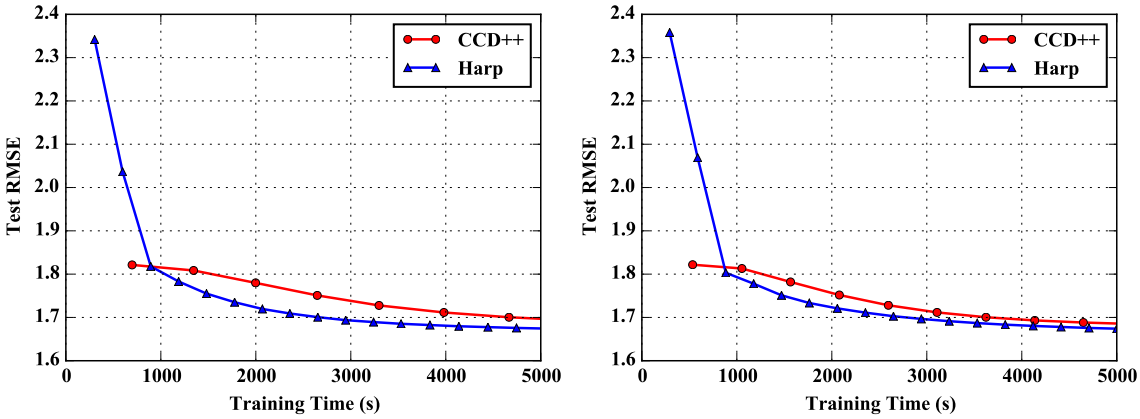


(d) “clueweb2”, Time to Convergence



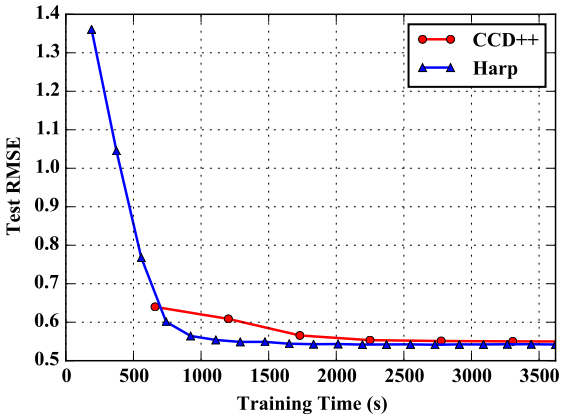
(e) “clueweb2”, Throughput Scalability

Figure 24: SGD for MF Model Convergence Speed

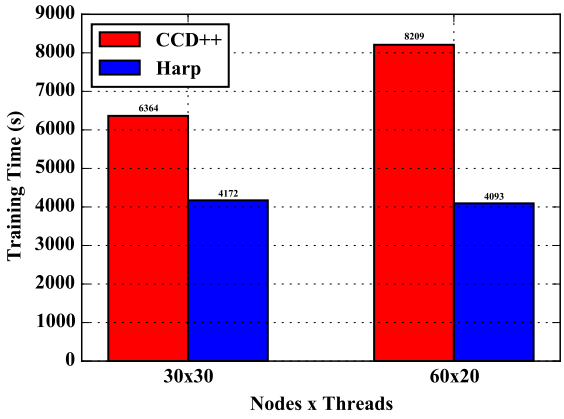


(a) CCD, “clueweb2”, 30x30

(b) CCD, “clueweb2”, 60x20



(c) CCD, “hugewiki”, 10x30



(d) CCD, “clueweb2”

Figure 25: CCD for MF Model Convergence Speed

9 Conclusion

In iterative machine learning algorithms, understanding the computation dependency between model updates is indispensable in creating efficient implementations. Through surveying the related research work, I identified four computation models to help users understand the mechanisms of model synchronization in machine learning parallelization. My research also indicates the advantages of collective communication techniques in implementing parallel big data machine learning algorithms. Therefore, I designed a MapCollective programming model for the big data software stack to enable model-centric computation with collective communication-based model synchronization. In the MapCollective model, Key-Value pairs are still used as the input data, and partitioned distributed datasets are used as the data abstraction to perform collective communication operations.

To test these ideas in practice, I created the Harp framework. Experiments on parallel machine learning applications show that the current Harp parallelization achieves high performance and is comparable to other leading implementations in the domain. The current Harp framework has been released at `https://dsc-spidal.github.io/harp/`. Through the efforts of other contributors, we have implemented and made available several machine learning applications, including K-means Clustering, Latent Dirichlet Allocation, Matrix Factorization, Multi-class Logistic Regression, and Support Vector Machine.

Future research directions include enhancing the reliability of the Harp framework, improving the expressiveness of the programming model, utilizing new hardware, and expanding the applicability to other machine learning applications. For the Harp framework itself, I did not focus on the fault tolerance in the current implementation. Since the execution flow in the MapCollective programming model is more flexible compared with existing MapReduce or graph programming models, it is a challenge to recover the computation automatically. However, check-pointing-based fault tolerance techniques should be investigated to guide developers to recover the computation in a simple way. The expressiveness of the Harp MapCollective programming model can also be improved so that developers can simplify their code for implementing machine learning applications. Ideally, developers hope that their sequential code can be simply parallelized with high performance. However, in reality, high-performance parallelization cannot happen without being

given detailed instructions. Following the guidelines in my thesis, developers can provide answers to the four questions of "what", "when", "where", and "how" in the model synchronization of a specific machine learning application. The challenge is if the development cost of parallel machine learning algorithms can be further reduced while preserving the freedom of conducting fine-grained performance optimization in the parallelization.

Other challenges of applying the Harp framework come from the algorithms and the hardware. Deep Learning algorithms have been parallelized by GPU with HPC methods applied [51], but the current Harp framework has yet to cover them. Though Java can improve the programming productivity in developing machine learning applications, it is inconvenient to operate high-performance computing devices and communication devices. Thus Java becomes a performance bottleneck to the Harp framework. One solution is to build a hybrid framework which uses the Harp collective communication operations for model synchronization, but leaves the local computation for high-performance machine learning libraries such as Intel DAAL[21]. Another option is to implement the methodologies in the Harp framework on top of Yarn MPI[22] with high level programming interfaces, which is able to provide both full access to high-performance hardware and convergence between the HPC and Big Data domains.

[21] https://software.intel.com/en-us/intel-daal
[22] https://software.intel.com/en-us/intel-mpi-library

References

[1] T. L. Griffiths and M. Steyvers, "Finding Scientific Topics," *PNAS*, 2004.

[2] P. Resnik and E. Hardist, "Gibbs Sampling for the Uninitiated," University of Maryland, Tech. Rep., 2010.

[3] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent Dirichlet Allocation," *JMLR*, 2003.

[4] D. W. Walker and J. J. Dongarra, "MPI: A Standard Message Passing Interface," in *Supercomputer*, 1996.

[5] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *CACM*, 2008.

[6] C.-T. Chu *et al.*, "Map-Reduce for Machine Learning on Multicore," in *NIPS*, 2007.

[7] J. Ekanayake *et al.*, "Twister: A Runtime for Iterative MapReduce," in *HPDC*, 2010.

[8] M. Zaharia *et al.*, "Spark: Cluster Computing with Working Sets," in *HotCloud*, 2010.

[9] G. Malewicz *et al.*, "Pregel: A System for Large-Scale Graph Processing," in *SIGMOD*, 2010.

[10] S. Kamburugamuve, "Survey of Apache Big Data Stack," Indiana University, Tech. Rep., 2013.

[11] B. Zhang, Y. Ruan, and J. Qiu, "Harp: Collective Communication on Hadoop," in *IC2E*, 2015.

[12] S. Lloyd, "Least Squares Quantization in PCM," *Information Theory, IEEE Transactions on*, vol. 28, no. 2, pp. 129–137, 1982.

[13] Y. Ruan and G. Fox, "A Robust and Scalable Solution for Interpolative Multidimensional Scaling with Weighting," in *e-Science*, 2013.

[14] Y. Koren *et al.*, "Matrix Factorization Techniques for Recommender Systems," *Computer*, 2009.

[15] H.-F. Yu *et al.*, "Scalable Coordinate Descent Approaches to Parallel Matrix Factorization for Recommender Systems," in *ICDM*, 2012.

[16] J. Qiu and B. Zhang, "Mammoth Data in the Cloud: Clustering Social Images," in *Clouds, Grids and Big Data*, ser. Advances in Parallel Computing, 2012.

[17] B. Zhang and J. Qiu, "High Performance Clustering of Social Images in a Map-Collective Programming Model," in *SoCC*, 2013.

[18] B. Zhang, "A Collective Communication Layer for the Software Stack of Big Data Analytics," in *IC2E Doctoral Symposium*, 2016.

[19] B. Zhang, B. Peng, and J. Qiu, "High Performance LDA through Collective Model Communication Optimization," in *ICCS*, 2016.

[20] B. Zhang, B. Peng, and J. Qiu, "Model-Centric Computation Abstractions in Machine Learning Applications," in *BeyondMR*, 2016.

[21] B. Zhang, B. Peng, and J. Qiu, "Parallelizing Big Data Machine Learning Applications with Model Rotation," ser. Advances in Parallel Computing, 2017.

[22] L. Bottou, "Large-Scale Machine Learning with Stochastic Gradient Descent," in *COMPSTAT*, 2010.

[23] Q. Ho *et al.*, "More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server," in *NIPS*, 2013.

[24] M. Chowdhury *et al.*, "Managing Data Transfers in Computer Clusters with Orchestra," in *SIGCOMM*, 2011.

[25] M. Li *et al.*, "Scaling Distributed Machine Learning with the Parameter Server," in *OSDI*, 2014.

[26] A. Smola and S. Narayanamurthy, "An Architecture for Parallel Topic Models," *VLDB*, 2010.

[27] A. Ahmed *et al.*, "Scalable Inference in Latent Variable Models," in *WSDM*, 2012.

[28] S. Lee *et al.*, "On Model Parallelization and Scheduling Strategies for Distributed Machine Learning," in *NIPS*, 2014.

[29] E. P. Xing *et al.*, "Petuum: A New Platform for Distributed Machine Learning on Big Data," *IEEE Trans. Big Data*, 2015.

[30] J. K. Kim *et al.*, "STRADS: A Distributed Framework for Scheduled Model Parallel Machine Learning," in *EuroSys*, 2016.

[31] L. Yao, D. Mimno, and A. McCallum, "Efficient Methods for Topic Model Inference on Streaming Document Collections," in *KDD*, 2009.

[32] Y. Wang *et al.*, "PLDA: Parallel Latent Dirichlet Allocation for Large-Scale Applications," in *AAIM*, 2009.

[33] J. E. Gonzalez *et al.*, "Powergraph: Distributed Graph-Parallel Computation on Natural Graphs," in *OSDI*, 2012.

[34] Y. Wang *et al.*, "Peacock: Learning Long-Tail Topic Features for Industrial Applications," *ACM TIST*, 2015.

[35] D. Newman *et al.*, "Distributed Algorithms for Topic Models," *JMLR*, 2009.

[36] E. Chan *et al.*, "Collective Communication: Theory, Practice, and Rxperience," *Concurrency and Computation: Practice and Experience*, 2007.

[37] Y. Low *et al.*, "Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud," *VLDB*, vol. 5, no. 8, pp. 716–727, 2012.

[38] Y. Bu *et al.*, "HaLoop: Efficient Iterative Data Processing on Large Clusters," *VLDB*, vol. 3, no. 1-2, pp. 285–296, 2010.

[39] R. S. Xin *et al.*, "GraphX: A Resilient Distributed Graph System on Spark," in *GRADES*, 2013.

[40] T. Gunarathne, J. Qiu, and D. Gannon, "Towards a Collective Layer in the Big Data Stack," in *CCGrid*, 2014.

[41] D. Sculley, "Web-Scale K-Means Clustering," in *WWW*, 2010.

[42] G. Fox, "Robust Scalable Visualized Clustering in Vector and non Vector Semimetric Spaces," *Parallel Processing Letters*, vol. 23, no. 02, 2013.

[43] Y. Ruan *et al.*, "Integration of Clustering and Multidimensional Scaling to Determine Phylogenetic Trees as Spherical Phylograms visualized in 3 Dimensions," in *CCGrid*, 2014.

[44] F. Yan, N. Xu, and Y. Qi, "Parallel Inference for Latent Dirichlet Allocation on Graphics Processing Units," in *NIPS*, 2009.

[45] C. Teflioudi, F. Makari, and R. Gemulla, "Distributed Matrix Completion," in *ICDM*, 2012.

[46] H. Yun *et al.*, "NOMAD: Non-locking, stOchastic Multi-machine algorithm for Asynchronous and Decentralized matrix completion," *VLDB*, 2014.

[47] Y. Zhuang *et al.*, "A Fast Parallel SGD for Matrix Factorization in Shared Memory Systems," in *RecSys*, 2013.

[48] B. Recht *et al.*, "HOGWILD!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent," in *NIPS*, 2011.

[49] R. Gemulla *et al.*, "Large-Scale Matrix Factorization with Distributed Stochastic Gradient Descent," in *SIGKDD*, 2011.

[50] R. A. Levine and G. Casella, "Optimizing Random Scan Gibbs Samplers," *JMVA*, 2006.

[51] A. Gibiansky, "Bringing HPC Techniques to Deep Learning," Baidu Research, Tech. Rep., 2017, http://research.baidu.com/bringing-hpc-techniques-deep-learning/.

Bingjing Zhang

EDUCATION

School of Informatics and Computing, Indiana University, Bloomington, IN
Ph.D. of Science in Computer Science — August 2009 - May 2017
GPA: 3.98/4.0
Master of Science in Computer Science — August 2009 - June 2011
GPA: 3.94/4.0
Software Institute, Nanjing University, Nanjing, China
Master of Engineering in Software Engineering — September 2007 - June 2009
GPA: 92.5/100
Bachelor of Engineering in Software Engineering — September 2003 - June 2007
GPA: 86/100

WORK EXPERIENCE

Microsoft Research, Redmond, WA
Research Intern — June 2 - August 22, 2014
- Worked on Retainable Evaluator Execution Framework (now Apache REEF)

Facebook, Menlo Park, CA
Software Engineer Intern — May 20 - August 9, 2013
- Algorithmic message reduction in Apache Giraph

School of Informatics and Computing, Indiana University, Bloomington, IN
Volunteer in Science Cloud Summer School — July 30 - August 3, 2012
- Prepared slides for the presentation Data Mining with Twister Iterative MapReduce
- Answered questions in hands-on session

Mentor in IU Summer Research Internships in Informatics and Computing Program — June - July, 2012
- Taught students how to use Twister iterative MapReduce framework

Teaching Assistant for A110: Introduction to Computers & Computing — September - December, 2009
- Held lab sessions and help desk hours

IBM China Research Laboratory, Beijing, China
Intern — September 2008 - June 2009
- Developed the web-page based control panel for a virtual machine management platform (programming in JavaScript, PHP)
- Developed a tool (programming in Python) to analyze the scalability of the virtual machine management platform.

TECHNICAL SKILLS

Java, Hadoop, Collective Communication, Parallel Computing, Machine Learning

TESTS & CERTIFICATIONS

IBM Certified Database Associate-DB2 Universal Database V8.1 Family	2005
IBM Certified for On Demand Business-Solution Designer	2005

HONORS & AWARDS

Excellent Graduate Student, Nanjing University	2008
Tung OOCL Scholarship, Nanjing University	2008
Outstanding Graduate, Nanjing University	2007
People's Scholarship I, Nanjing University	2006
People's Scholarship III, Nanjing University	2005
People's Scholarship II, Nanjing University	2004

RESEARCH EXPERIENCE

School of Informatics and Computing, Indiana University, Bloomington, IN

Research Assistant — January 2010 - present

- Developed Harp, a machining learning framework with a collective communication library on Hadoop for big data (programming in Java)

PUBLICATIONS

1. **B. Zhang**, B. Peng, J. Qiu. Parallelizing Big Data Machine Learning Applications with Model Rotation. Book Chapter in series Advances in Parallel Computing, 2017.
2. **B. Zhang**, B. Peng, J. Qiu. Model-Centric Computation Abstractions in Machine Learning Applications. Extended Abstract in BeyondMR, 2016.
3. **B. Zhang**, B. Peng, J. Qiu. High Performance LDA through Collective Model Communication Optimization. ICCS, 2016.
4. **B. Zhang**. A Collective Communication Layer for the Software Stack of Big Data Analytics. IC2E Doctoral Symposium, 2016.
5. **B. Zhang**, Y. Ruan, J. Qiu. Harp: Collective Communication on Hadoop. Short Paper in IC2E, 2015.
6. **B. Zhang**, J. Qiu. High Performance Clustering of Social Images in a Map-Collective Programming Model. Poster in SoCC, 2013.
7. J. Qiu, **B. Zhang**. Mammoth Data in the Cloud: Clustering Social Images. Book chapter in Cloud Computing and Big Data, series Advances in Parallel Computing, 2013.
8. T. Gunarathne, **B. Zhang**, T.-L. Wu, J. Qiu. Scalable Parallel Computing on Clouds using Twister4Azure Iterative MapReduce. Future Generation Computer Systems, 2013.
9. T. Gunarathne, **B. Zhang**, T.-L. Wu, J. Qiu. Portable Parallel Programming on Cloud and HPC: Scientific Applications of Twister4Azure. UCC, 2011.
10. **B. Zhang**, Y. Ruan, T.-L. Wu, J. Qiu, A. Hughes, G. Fox. Applying Twister to Scientific Applications. CloudCom, 2010.
11. J. Qiu, J. Ekanayake, T. Gunarathne, J. Y. Choi, S.-H. Bae, H. Li, **B. Zhang**, T.-L. Wu, Y. Ruan, S. Ekanayake, A. Hughes, G. Fox. Hybrid cloud and cluster computing paradigms for life science applications. BMC Bioinformatics, 2010.
12. J. Ekanayake, H. Li, **B. Zhang**, T. Gunarathne, S.-H. Bae, J. Qiu, G. Fox. Twister: A Runtime for Iterative MapReduce. Workshop on MapReduce and its Applications, HPDC, 2010.